Il6r
no. 618 UIUCDCS-R-75-618

MULTI-BATCH: A MULTIPROGRAMMED MULTIPROCESSOR BATCH SYSTEM FOR THE PDP-11 COMPUTER

by

Gregory Lawrence Chesson

April 1975

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-75-618

MULTI-BATCH:
A MULTIPROGRAMMED MULTIPROCESSOR BATCH SYSTEM FOR THE PDP-11 COMPUTER

by

Gregory Lawrence Chesson

April 1975

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

This work was supported in part by the National Science Foundation under Grant No. US NSF DCR72-03740A1 and was submitted in partial fulfillment for the Master of Science degree in Computer Science, 1975.

ACKNOWLEDGMENT

I am grateful to Professor Donald B. Gillies for his support and encouragement. I also wish to acknowledge Professors Gillies, J. W. S. Liu, C. L. Liu, and M. D. Mickunas for their collective wisdom and advice.

Special thanks are due to persons whose assistance was invaluable. These include graduate students Elmer McClary, Tom Miller, and Ian Stocks; draftsmen Stan Zundo and Randy Bright; and a few hundred undergraduate students who became Multi-Batch users. I am particularly indebted to the Department of Computer Science for providing facilities and a congenial working environment.

TABLE OF CONTENTS

1. INTRODUCTION

1.0 Exposition

Multi-Batch is a software system for the PDP-11 computer. It is the product of two separate projects that were merged into one. The first project involved adding HASP-like [2] spooling capabilities to the University of Illinois PDP-11 BATCH job monitor [7]. The second project involved a study of methods for implementing interprocess communication in multiprocessor systems. The merger of these two projects resulted in a batch monitor which can operate a "foreground" job stream and a number of "background" spooling processes. Communication between Multi-Batch processes is implemented by IPC (interprocess communication) software, and the background processes are designed to be distributed among processors in a multiprocessor environment.

This paper is organized into three chapters. The remainder of this chapter will serve as an overview of the system. This will take the form of a chronological description of the heretofore undocumented software that comprises BATCH, and which is incorporated into Multi-Batch. Chapter 2 presents general aspects of the system design of Multi-Batch. The main topics are multiprogramming, interprocess communication, and system reliability. Chapter 3 gives detailed operational information on the data and control flows in the system.

1.1 Software Genealogy of Multi-Batch

Multi-Batch depends on a modified version of DEC's DOS operating system for the PDP-11. DOS was acquired by the Digital Computer Laboratory in 1971. The debugging and improvement of this primitive I/O monitor and file system continues to this day.

In the early months of 1971 Russel Atkinson developed card reader and line printer drivers for DOS. He also added a software routine called KBT to the DOS teletype driver. When enabled by a special call through the DOS I/O system, KBT reads input data from a disk file on behalf of the normal teletype device driver. Disk files used in this manner normally contain a number of command strings which are executed sequentially by DOS. The DOS command files were named Zip files after a similar facility with the same name that was observed on Burroughs' [3] systems. Later enhancement to DOS Zip file processing permitted argument substitution in command lines similar to the exec_com facility in Multics [4] or "indirect" files in DEC's RSX11/m system [5]. However, DOS command files are not capable of conditionally altering the flow of control as are the Multics and RSX implementations. Zip files remain sequential in nature like catalogued "proc's" under OS/360 [6].

Working concurrently with Atkinson, Roger Haskin developed a simple Batch job monitor using the Zip-file mechanism. The normal mode of operation of that first system was to read the cards for the next job, print the listings generated by the previous job, and when both of these tasks were completed initiate a Zip file containing the appropriate JCL for the next job. The last command in the Zip file would once again invoke the Batch monitor.

Because the PDP-11 used for this work did not have memory management hardware, the core-resident portion of the operating system was vulnerable to programs containing addressing errors. Since the PDP-11 was primarily used at that time by assembly language programming classes, one would expect a high incidence of programs with addressing errors. Atkinson addressed this problem by implementing an interpreter for PDP-11 object code which supervized the execution of student jobs. The interpreter protected the operating system from malfunctioning programs by preventing memory access outside of the program's data and stack space, by checking calls to the operating system for validity, by providing simplified I/O calls for the user, and by enforcing I/O and time limits. The interpreter also provided trace, dump, and diagnostic information which simplified debugging chores for student programmers.

About six months after the first Batch monitor and interpreter were developed, Ian Stocks introduced several improvements. The first change involved partitioning the Batch monitor into 3 coroutines (CDSK, DLP, and KBSH) and a startup routine (MAINBA). CDSK spooled cards to disk. DLP spooled disk files to the printer. KBSH was a keyboard shell that handled various teletype commands. The effect of the three coroutines was a continual "polling" of the I/O devices, leading to a high degree of I/O overlap during spooling operations. The apparent "simultaneous" operation of I/O devices was an improvement over the previous Batch system. Stocks also added special error trapping returns to the card reader and line printer drivers. This allowed the Batch error routines to recover from temporary errors such as card jams or printer paper outages without interrupting the operation of error-free devices. The error handling in Batch required no teletype communication with an operator, and would restart a device when the error was cleared. This scheme is preferable to the standard DOS error handling which requires a teletype response and which suspends the system until a response is received.

In the spring of 1973, Mike Selander wrote a program called MAKER that simplified the production of special Zip files, or "proc's", for Batch. These proc's were regular Zip-files with some additional preface information. The preface information was used by Batch to set up files for the proc, and the Zip part of the file was passed to DOS for execution. In practice, the first card of a student job was interpreted as the name of a catalogued proc. Assuming the proc was catalogued and correct, Batch would attempt to build input files from the card reader according to the preface information in the proc. While building input files, Batch would print output files according to preface information found in the just-completed proc. When both spooling operations were done, Batch would initiate the Zip file part of the current proc and exit. The operation of exiting would force DOS to read a command from the teletype. Since the Zip option was on, succeeding commands would come from the Zip file until Batch was again invoked by a "Run Batch" command in the file.

Various improvements were made to Batch and to the the interpreter by Dave Kassel [7] in 1973. These changes included the addition to KBSH of "restart" and "kill" commands for listings, and the addition of a backup mechanism to Batch's job counting routine. The backup mechanism stored the current job count in the bootstrap area of the system disk from whence it could normally be restored after a system crash or reboot.

Until October of 1973 software work on Batch had consisted of minimal maintenance activity. At that time Multi-Batch development commenced with the successful demonstration by the author and Elmer McClary of a simple method for adding "background" processes to DOS. By December of 1973 McClary had produced the first version of a background line printer process. By the middle of January 1974 the author had produced startup routines, queuing software, a modified version of the line printer process, and a modified DLP for Batch. This version of Batch attempted to spool line printer output continuously while proc's were executing. The system was unreliable until a DOS error found by Ian Stocks proved to be the principal culprit. During the spring of 1974 the IPC scheme was designed and implemented.

The system design of Multi-Batch was completed in the spring of 1974, and the initial implementation was carried out by the author and Elmer McClary in about two months time during the summer of 1974. Attempts to operate Multi-Batch in the fall of 1974 as a production system for student programs revealed numerous flaws. By the end of November the system had been made reliable enought to support continuous student usage. Although multiple-processor operation of the IPC has not been demonstrated on a production system, the documentation presented in the following pages will remain the system blueprint for continued development.

2. SYSTEM OVERVIEW

2.0 Multiprograming

Multi-Batch is a "foreground/background" type of system in which the background process is "interrupt-driven." The system operates the disk-to-printer spooling process and the card-to-disk process in "background" while the batch job stream is executed in "foreground." Each background process is active only when serving a hardware or software interrupt. That is, Multi-Batch does not time-slice the central processor. For example, when the line printer process is started, the foreground software passes to it the disk address of a print file. The line printer program initiates a disk read to obtain the first page of text and then exits, effectively going to sleep until the disk interrupt signaling I/O completion occurs. When the print program is interrupted by the disk completion return, the data from disk is sent to the line printer, and the print process is again suspended until the line printer driver interrupts with a printer completion return. This process "see-saws" continuously between waiting for a disk transfer to complete and waiting for the printer to finish with a data buffer.

The time required to process a completion return and initiate the next I/O operation is 400 microseconds or less on the PDP-11/20. This overhead is incurred about once per second when driving a 600 line per minute printer. One concludes that the CPU overhead of the line printer process is negligible. The card reader process is more complicated than the printer process because several data structures must be maintained in addition to controlling the drivers. However, the input process overhead is almost as "invisible" as the line printer process.

There is no need to time-slice the CPU in Multi-Batch because the background segment is interrupt-driven as explained above. Hence the DOS modifications needed to support Multi-Batch are minimized. In fact, the only DOS services used by the background segment are the device drivers, and they are called at the system-driver interface level to avoid the DOS supervisory I/O routines. Thus DOS is completely unaware of the activities of the background segment. The isolation of the background segment from DOS and foreground programs is further illustrated in Figure 2.0.0. Figure 2.0.0a depicts the normal utilization of memory by DOS. Note that RMON, the resident part of the DOS monitor, occupies the low area of memory, and the user program is located in the high area of memory. The stack and system buffer area grow toward each other. DOS's core allocator uses the bitmap to keep track of free space in the buffer area. Figure 2.0.0b shows where the background segment is located when Multi-Batch is running, i.e. it is essentially made part of RMON.

The functional autonomy of the Multi-Batch background routines with respect to DOS is actually a design goal, rather than an accident of the implementation process. Any other scheme for adding concurrent processes to a single-user monitor like DOS must subvert that monitor's internal control procedures, as well as provide additional ones. The experience of this author has been that it is usually difficult, and often ill-advised, to modify existing software to perform tasks that were never

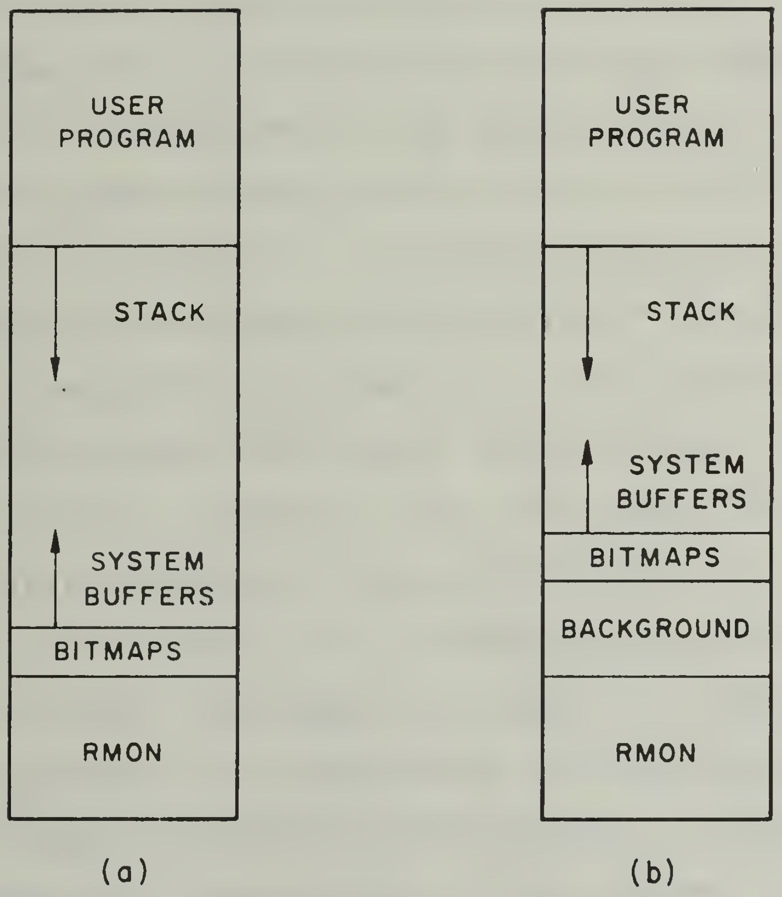


Figure 2.0.0

DOS Memory Map

expected of the original design. As a case in point, the author and Tom Miller produced a working version of a time-sliced DOS system during the summer of 1973. Although this system did provide up to four "virtual" copies of DOS, a reliable production version of this system would have required extensive revisions to DOS. It was concluded that a new operating system could be written in less time than that required to rewrite DOS. Therefore the time-slicing system died a natural death in deference to the autonomous background scheme.

The job of loading the background segment into core and submerging it in RMON is done by the background initiator program (BKINIT). BKINIT modifies several key parts of DOS when loading the background segment. These key parts are saved in a disk file (STATUS.COM) before being overwritten so that the background segment can be extracted cleanly. BKINIT is designed to be table-driven by network information inserted in STATUS.COM by the configuration program (CON).

In theory, one can reconfigure the system by running CON, giving CON configuration information via teletype, and then running BKINIT. Configuration information includes such things as the number of processors in the network, and characteristics of the interprocessor connections. In the present system, the configuration information is stored in a file called BATCH.BAT. A student user can bootstrap the system by pressing front panel switches on the CPU, and then typing "BATCH" on the console.

This will start up the Zip-file, BATCH.BAT, which runs CON and BKINIT with preset input parameters, followed by the Multi-Batch job monitor, BATCH.LDA.

2.1 Interprocess Communication

2.1.0 General Comments

The term "interprocess communication," or IPC, denotes the transfer of data between cooperating processes in a computer system. Communication between processes is a fundamental issue of distributed system design. Although it is also an important consideration in more traditional systems, a unified approach to communication is absolutely essential in distributed software. Postel's report [22] relates IPC facilities to the development of distributed software, pointing out that the lack of uniform access to ARPA network IPC facilities accounts for the small amount of network software that has been developed. The importance of an IPC is further emphasized in a system like RSEXEC [20] which depends on a network IPC for many operations.

In an effort to obtain a clearer understanding of distributed systems, we studied many existing IPC designs - both for distributed and non-distributed systems. A number of systems [11,12] provide IPC services that resemble ordinary I/O operaions, e.g. Read and Write. Other systems [3,4,13,15,23,24] provide process control facilities which augment or relate to the IPC service. Still others [16,18,21,24] provide FIFO message queues for IPC traffic. And at least one system [17] defines communication between virtual machines.

Although various IPC schemes seem to differ in many ways, the essential differences are related to the notion of control. By control, we mean the notification of arrival that accompanies the transmission of IPC data. The mechanism may be implicit, as in a read or a write command where a process waits until the operation finishes. It may also be explicit, as in systems which provide software interrupt facilities.

In designing an IPC scheme for Multi-Batch, we want to provide a set of primitive IPC operators from which all other reasonable communication disciplines could be synthesized. As it happens, a single primitive operation will suffice. This operation is implemented in Multi-Batch by the CALL macro, described here in section 2.1.3. After some additional preliminary comments we will discuss IPC processing funcions (section 2.1.1), particulars of the programmer's interface to the Multi-Batch IPC (sections 2.1.2 and 2.1.3), and finally the internal organization of our prototype system (section 2.1.4).

Multi-Batch consists of several modules. A module contains one or more procedures. A system of modules is treated as a system of parallel processes. A program that cannot be realized meaningfully as a system of modules would be coded as a single module. (Experience shows that many programs, and virtually all large systems of programs, decompose naturally into parallel processes. Baer [8,9] describes techniques that can be applied in the decomposition process. Bowie [10] applies such techniques to achieve a parallel decomposition of OS/360.)

One important consideration of our IPC design was that a system of modules should be adaptable to changes in hardware configuration without any software changes. Hence a system could be implemented and checked out using only one processor, while production versions could be distributed among several processors. In addition, the production program could be run on a variable number of processors depending on the availability of or need for network resources. The Multi-Batch IPC meets this requirement via naming conventions used in the CALL macro. That is, the CALL macro requests communication between processes using only the programmer-defined names of the communicating elements. The IPC software translates these names into physical communication paths that are tranparent to software modules. Thus, as long as each module is programmed with an understanding of the timing variations that might occur as a result of different distributions of modules to processors, a system of modules can achieve a certain degree of independence with respect to processor resources. At least one computer manufacturer [12] is developing systems that incorporate some of these ideas. The referenced system provides reasonably versatile communication and process control facilities, although it is not yet oriented toward the concept of treating a system of parallel tasks as a unit.

The macro statements described in section 2.1.2 and 2.1.3 resemble the programming language statements that were described in another paper by this author [1]. Differences that exist between the system proposed in [1] and Multi-Batch result from implementation expediencies adopted by the implementors. However, the semantics of the CALL statement in [1] are emulated exactly by the CALL macro in Multi-Batch.

2.1.1 IPC Functions

In this section we define the processing functions that are associated with interprocess communication in a computer network. These functions are represented by the ellipses in figure 2.1.0. We will refer to the collection of functions in the drawing as IPC "support software", or the IPC "kernel."

CHANNEL DRIVERS are those procedures in an IPC kernel which control the physical operation of processor-to-processor data They may be implemented by shared memories, telephone connections, digital bussing, and other techniques. Each technique usually requires a unique device-handling and signaling protocol. Hence it is desireable that differences between data channels be visible only to the channel drivers so that software using the channels can use standard I/O calls on the drivers. Channel drivers designed for Multi-Batch perform checksum tests on incoming data and automatically request retransmission when error are found. These drivers use a write-demand algorithm. The essence of the write demand scheme is that channel drivers receive both read and write requests from local software, but send only the write commands to each other. A write command sent through a channel will be acknowledged by the driver on the other end when the other driver is ready to receive. This will happen when the other driver has a matching read command. Read commands are allowed to languish in the driver's "read list" until a matching write command arrives. Similarly, unmatched write commands are kept in a "write list." In our scheme, all driver

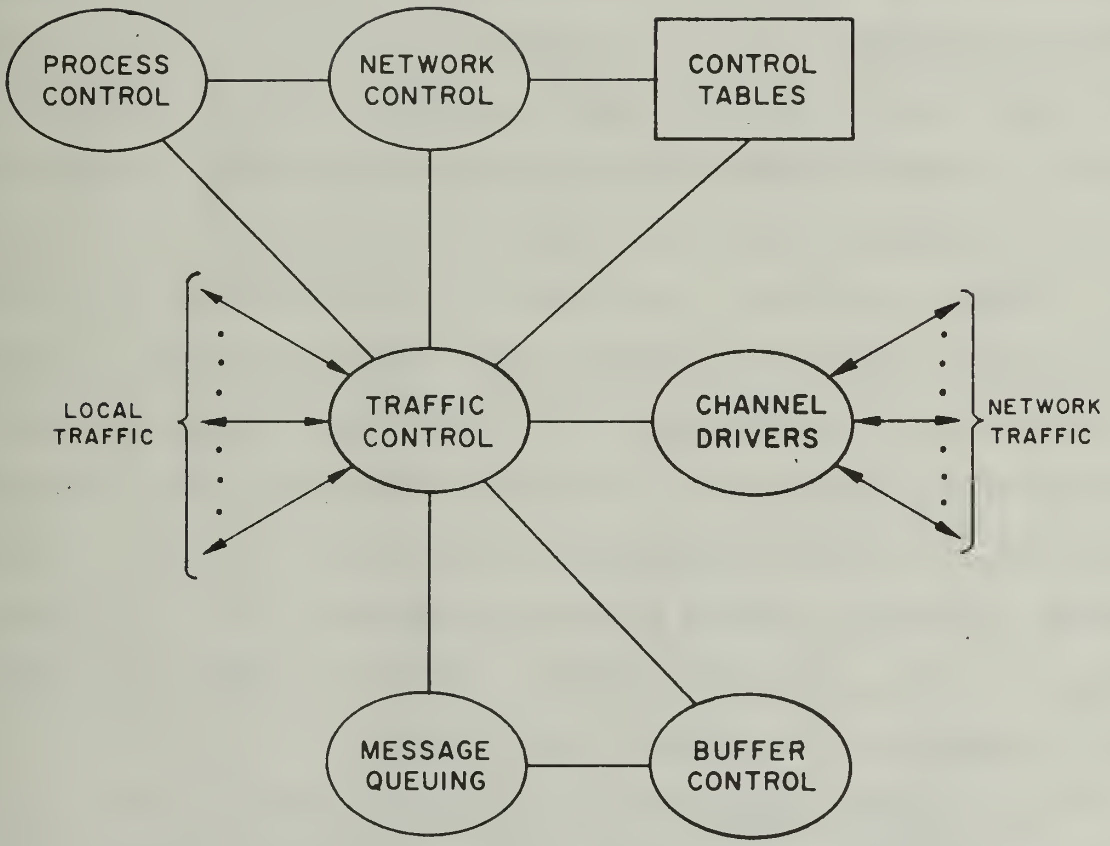


Figure 2.1.0

IPC Functions

commands include a one word process identifier (PID) and the total number of words requested for transfer. Thus the definition of a "match" between read and write commands is that they must agree with respect to message identifiers and transfer length. Although it is useful in some applications to be able to transfer only a part of an I/O request, this was not a criterion in Multi-Batch.

Note that the write-demand philosophy is applied at a low level in the hierarchy of IPC communication, and that "read" and "write" requests from a user process (high-level) are translated by system software into write demands at the channel level. Read requests at the channel level are sent only by kernel processes that are expecting traffic. For computers that are in close enough proximity so that the usual full handshaking protocols are not necessary for reliability, the write-demand scheme affords a minimum of control overhead on the channel as well as in the driver without any loss in programming flexibility.

We can now identify a hierarchy of controls associated with a channel. At the lowest level (level-0) are the basic device-handling protocols. The next higher control level (level-1) includes the I/O requests and data transfers that are part of the write-demand scheme outlined above. Level-1 transactions are generated by control algorithms that direct traffic through the channels, i.e. the traffic control which is discussed below. At least one more level (level-2) of communication can be distinguished. This level is associated with IPC requests generated from user modules. Arbitrarily many additional levels can be defined above level-2 in accord with the system hierarchies that use the IPC mechanism. However, it will suffice for this discussion to restrict our attention to these three areas of activity.

TRAFFIC CONTROL refers to that part of the kernel which translates IPC data transfer requests into data transfer commands to the rest of the system. Using the terminology introduced above, the traffic control algorithm is a mapping between level-2 and level-1. Referring to figure 2.1.0, level-2 is represented by the part labeled "local traffic." Level-1 activity is represented by the line that connects TRAFFIC CONTROL and CHANNEL DRIVERS. The algorithms needed to accomplish the level-2/level-1 mapping are basically routing procedures, although sophisticated systems may introduce priority queues and other side-effects into the mapping. The archetypal routing scheme accesses a tabular data structure using a process name as the key. The selected table element would reveal the location of a process in the system as well as the values of state variables used to monitor relations between processes. State variables might indicate such things as whether a process is "busy", or not able to receive messages, or that a message was sent to a process and no acknowledgement was received, etc. If an IPC implementation includes more than one data transfer mechanism, then we would also expect to see a state variable for each process indicating its current communication attributes. We use the term, CONTROL TABLES, to denote the composite routing and state information required by a traffic control algorithm.

MESSAGE QUEUING and BUFFER CONTROL are self-explanatory titles. Basically, the queuing facilities in an IPC kernel allow message traffic to accumulate at the level-2/level-1 boundary and the level-1/level-0 boundry. The queued items at the low level boundary are I/O requests. At the level-2/level-1 boundary, the queued items are IPC messages. Queuing at these two boundaries tends to smooth the traffic bottlenecks caused by disparities between different bandwidths and loadings of the message-handing processes and channels in a system.

BUFFER CONTROL is our term for the space management algorithms that support message queuing. In systems where IPC messages are restricted to small fixed-length structures [4,18], buffer control can be accomplished with relatively simple algorithms and a pool of system buffers. Support for variable message sizes [3,24] can be provided elegantly if the host operating system permits buffer allocation on the stack of a message-handling process.

PROCESS CONTROL denotes the part of a system that supervises the initiation, suspension, activation, and interruption of processes. This facility must be available to kernel routines that are supposed to influence the execution of processes using the IPC. For example, in the UNIX [13,14] operating system a process is allowed to send data to another process through a mechanism called a pipe. The system will buffer up to 4096 bytes of data at which point the sending process is suspended. A read operation on the pipe by the receiving process will reactivate the sending process. The pipe software communicates with the standard process controls in the UNIX kernel for the necessary suspension and activation functions. It is interesting to note that a number of systems [4,11,15] provide a special IPC for this kind of communication. In these systems it is possible to activate a process and send it a short (1-word) message. This facility is useful for synchronizing hardware interrupt routines with higher level processes.

NETWORK CONTROL refers to the allocation of resources in the distributed system. In a distributed IPC-based system, all resources (e.g. processors, files, devices) would be "owned" by software processes. Acquisition of resources must be accomplished by communicating with the owners. In figure 2.1.0, NETWORK CONTROL represents all the "owner" processes in one processor, as well as the appropriate allocation algorithms for those resources. The line from NETWORK CONTROL to CONTROL TABLES indicates that the allocation procedures should maintain state variables for processes in the local tables.

2.1.2 IPC Declarative Macro's: ENTRY, NAMER

Every module in our IPC scheme contains one or more procedures. The notion of procedure in this context is exactly the same as the notion of a procedure in a high-level language. However, since Multi-Batch is implemented in a macro assembly language, there is no formal mechanism for declaring procedures and their parameter lists. Instead, we declare only the entry point of each procedure. The ENTRY macro is used for this purpose. It must be placed at the beginning of a module, before any code-generating macro's or machine instructions. As shown below, entry labels are parameters to the ENTRY macro. Multiple labels are separated by commas within angle brackets. The angle brackets are required symbols, not meta-notation. It may be necessary to use the macro more than once since macro parameter lists are not permitted to extend past line boundaries in DEC's macro assembler. The form of the ENTRY macro is:

```
        ENTRY   <label-1, label-2,...>
```

The NAMER macro has the following form:

```
        NAMER   module,<entry list>
```

where "module" is a module name, and "entry list" is a list of entry points to that module. The NAMER macro's for all modules in Multi-Batch are stored in a file called NAMES.MAC. This file is appended to the front of each module when it is assembled. The NAMER macro's in the file equate each module and entry name to an integer. These integers become the indicies used by the traffic control to access entries in the control tables. The first module name encountered is assigned the integer 1. The second module is named 2, and so on. Similarly, the first entry name for each module is equated to 1, etc. The CALL macro takes module and entry names as arguments, but generates code that passes the integer equivalents to the IPC kernel. This eliminates name translation operations that would otherwise be required in the kernel.

2.1.3 IPC Control Macro's: EXIT, CALL

Much of the IPC activity in Multi-Batch occurs as a result of hardware interrupts caused by I/O devices. IPC communication itself is treated as a software interrupt. Since all processes in Multi-Batch are activiated by one or the other of these two kinds of interrupt, we use the term "suspend" to mean "release the processor and restore the software environment that existed prior to the interrupt." The macro EXIT is provided to suspend from IPC software interrupts. The form of the EXIT macro is:

```
        EXIT
```

The CALL macro provides the data transfer and software interrupt functions in the Multi-Batch IPC. It has the following form:

```
    CALL    <m1,e1><m2,e2>< parameter list >
```

where "m1" and "m2" are module names, "e1" and "e2" are entry points within m1 and m2, and the parameter list consists of variable names separated by commas. The module and entry names used in the CALL macro must appear in the NAMES.MAC header file (section 2.1.2). We first discuss the semantics of parameter passing, then the control mechanism of the CALL macro.

The variables in the parameter list are subject to certain conventions. That is, we distinguish two variable types: scalar, and composite. The class of scalar variables includes all constants or data types that are representable in two or less PDP-11 machine words. Thus, 32-bit integers and short (4-byte) strings would be considered scalars. All data types requiring more than two words of storage fall into the class of composite variables. This class would include long strings, arrays, and I/O buffers. Composite variables are specified in the parameter list by an <address,length> pair. In the current implementation, each scalar in a parameter list is represented by its name. A two-word scalar would require two names, viz. "x" and "x+2." Thus, a parameter list specifying 1-word scalar x, 2-word scalar y, and 256-word buffer z would appear as:

<x,y,y+2,<z,256>>

When the CALL macro is executed at run-time, it causes module m1 to be interrupted at entry e1. The scalar and composite variables are copied to a new location, possibly through a channel, where they can be accessed from m1. Thus, parameter-passing is of the "call-by-value" variety. If communicating modules share address space, then "call-by-reference" becomes meaningful, and descriptors which reference shared data areas can be passed by value.

The control mechanism implemented by the CALL macro has been proven useful for implementing the asynchronous control structures found in most operating systems, as exemplified to some degree by Multi-Batch. In essence, the CALL macro provides a method of initiating parallel procedures and establishing communication paths between them. In a distributed system where multiprogramming and multiprocessing are the norm, a construct such as the CALL described here can help synchronize the software. The following discussion assumes that a module, designated <m0,e0>, contains a CALL macro, and that the module and entry names appearing in the macro are <m1,e1> and <m2,e2> as in the example. For clarity we will also refer to <m0,e0> as TASK-0, <m1,e1> as TASK-1, and <m2,e2> as TASK-2.

When the CALL is executed, control is transferred from

<m0,e0> to COMM, the traffic control routine in Multi-Batch. COMM will locate <m1,e1> in the control tables and determine the proper course of action to take. The outcome of this will be that the current values of variables specified in the parameter list are copied to module m1 at entry point e1. Depending on whether <m1,e1> is in a quiescent state or actively executing, it will be either awakened or interrupted. As soon as the IPC message is started on the path to <m1,e1>, TASK-0 is allowed to continue execution at the instruction following the CALL. Thus when TASK-1 commences, we have two processes running in parallel. Each one is free to start other processes, or to suspend itself pending some system event. TASK-1 will usually be performing some computation for TASK-0, so the question arises of how the result of the computation will be communicated. Alternatively, TASK-0 and TASK-1 may be members of a linear array of processes such that <m0,e0> passes data to <m1,e1>, which in turn may operate on the data and pass it to another process. A case also arises where TASK-1 may be called from many different tasks, and is expected to notify one of possibly many "third parties" when its computation is complete. These are a few of the possibilities. The second process identifier, <m2,e2>, in the CALL macro accomodates each of these situations. When TASK-0 calls TASK-1, the identifier, <m2,e2>, is passed along with the calling parameters to TASK-1. This identifier can be used later by TASK-1 in a CALL macro that initiates return CALL's. A return CALL might be an acknowledgement of the initial CALL, or a signal indicating the completion of some processing task. The original

calling procedure, TASK-0 in the above, selects the target of the return CALL. It is this ability to control the return CALL that lends versatility the IPC.

We can speak of a CALL as having two "returns." The first return is to the statement following the CALL macro itself. This return is made by the IPC kernel upon successful initiation of the data transfer to the called process. The second return, or completion return, refers to the use of the argument, <m2,e2>. Normally, a completion CALL occurs at some time after the first return although this in not a general rule. For example, suppose that one process, P1, issues a CALL to a second process, P2, for I/O. The second process starts a transfer and suspends itself until the I/O completes. This allows the first process, P1, to resume. Eventually the I/O transfer will complete and cause a hardware interrupt. This interrupt will reactivate P2. At this time, P2 can use the return identifier to make a completion call. However, it happens in Multi-Batch that P2 may invoke the completion CALL without waiting for an I/O operation to finish, and without suspending itself. This means that P1 has not yet resumed execution. Thus P1 would be awakened by P2 before the first return.

Another case arises in which P1 may be communicating with processes other than P2. Then P1 may have need for a message queuing facility, or a means of blocking IPC interrupts when it is not prepared to receive messages. Neither of these techniques were required in Multi-Batch because all procedures that could experience this kind of situation were made reentrant.

2.1.4 IPC Data Structures

This section gives the essential details of the Multi-Batch IPC kernel. The discussion is organized around figures 2.1.1, 2.1.2, and 2.1.3, which depict system data structures, and figure 2.1.4 which shows the stack frames generated by the CALL macro and the kernel. We begin with figure 2.1.1.

Recall that module and entry names are represented as integers in the IPC kernel by virtue of the NAMER macro. These integer "names" are used to index directly into the routing and state tables. The base table is accessed with an index generated by the IPC kernel when modules are loaded for execution. We first consider the routing and state tables.

The routing table consists of n+1 1-word entries in consecutive memory locations, T[0] through T[n]. Each word, T[i], corresponds to a module (i.e. process) in the system such that the index, i, is the module "name." The correspondence also holds for the state table where entries contain state and attribute information for each process. In the routing table, T[i] will be zero if no module corresponding to that index is known to the system. Otherwise T[i] contains the memory address of a module, or the logical number of a channel through which T[i] can be reached. Since channels are numbered beginning with 1 and memory addresses of proccesses in Multi-Batch start above the resident monitor (RMON), there is no problem distinguishing between channel numbers and memory addresses. Also since all modules begin at an even address, an odd address is used to represent the situation where a process may be active but incoming IPC traffic must be processed in some special way. For example, we might look in the state table and discover that incoming messages must be added to a queue or sent to an alternate process.

A generalized IPC system must be able to support more than one set of modules at a time. If we use compiler-assigned or NAMER-assigned integer names for modules, naming conflicts will occur when the control tables are filled in at load time. An additional table, the BASE table, can resolve these conflicts. By assigning a "subsystem" number, k, to a module system when it is loaded, the table entries for each subsystem can be loaded into contiguous sections of the control tables with B[k] pointing to the first entry for each active subsystem, k. Figure 2.1.1 shows a table with three subsystems loaded. B[0], B[2], and B[3] mark the beginning of the table area for each subsystem. The present version of our IPC kernel need only service a single system of modules; namely the Multi-Batch modules. Therefore, the base table is omitted from this implementation.

Other table areas in figure 2.1.1 marked STATIC and DYNAMIC delimit areas reserved for permanent and transient communication paths. Static entries direct IPC traffic to standard system processes such as resource allocators, process control routines, and file system routines. The dynamic entries in the table provide for "one-shot" paths that may be established for a single message transfer. Dynamic entries might also be useful in a context where the traffic controller monitors IPC traffic and performs rerouting based on observed channel loadings.

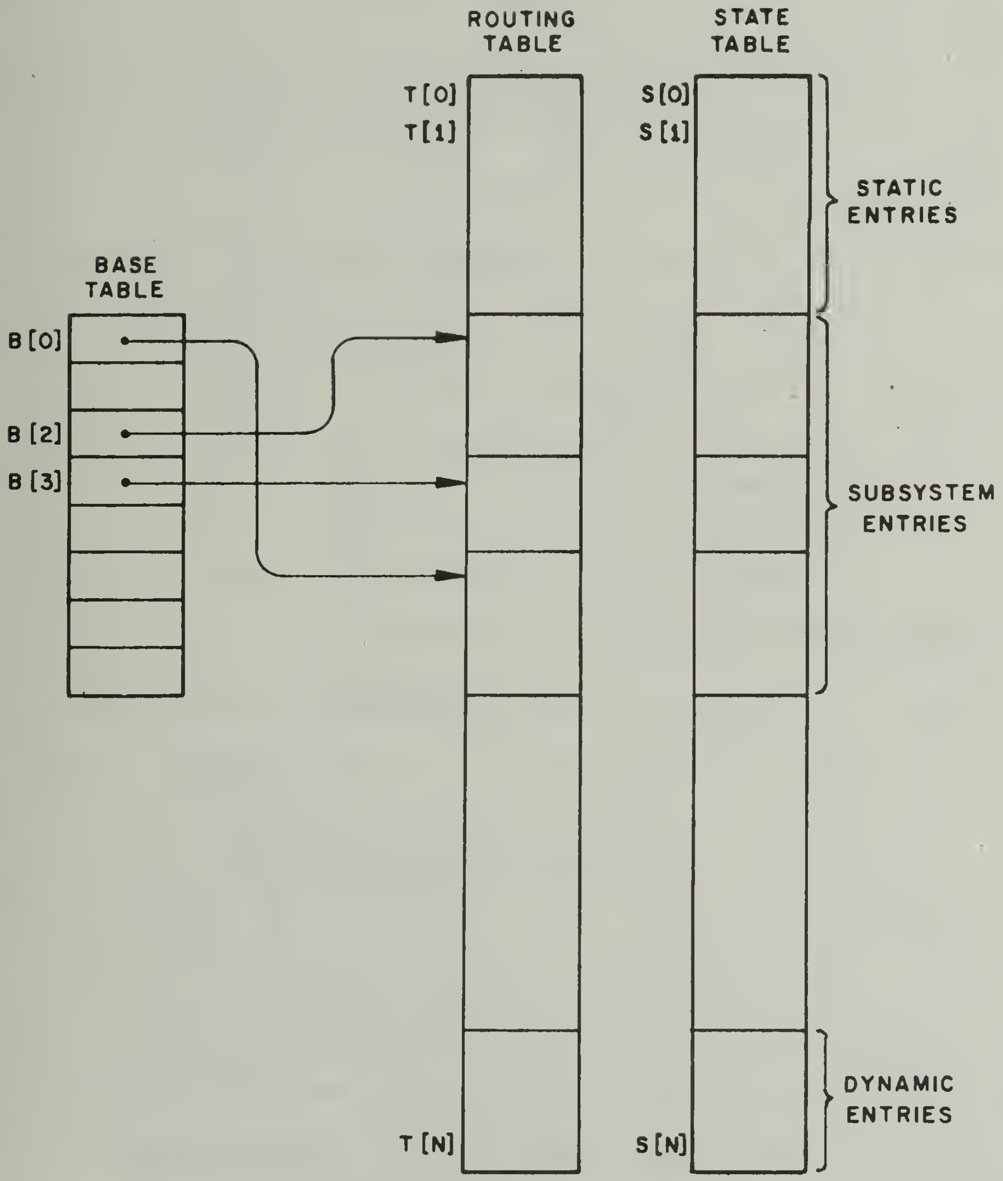


Figure 2.1.1

Network Control Tables

The physical location of the subsystem and dynamic entries in the table indicate the allocation strategy; viz. subsystem entries are allocated from one end of the table and dynamic entries are allocated from the other end. However, there were no requirements for temporary table entries in Multi-Batch. They are mentioned here for completeness only.

Figure 2.1.2 indicates a possible setting of three routing tables, RT1, RT2, and RT3, for a six module subsystem. The bold arrows indicate table entries that contain memory address. The other lines represent communication through channels. Thus module A occupies one processor; C,D, and F are on a second processor; and modules E and B are on the third. Note that each routing table gives a path to each of the six modules thus implementing a full-crossbar connection capability between modules. Note that certain paths are longer than others. For example, if F sent a message to D, it would be routed through RT1. A message from E to F would require on a single channel transmission. The decision process leading to a particular setting of the routing tables may depend on statistical monitoring of channel loadings, on the expected IPC behavior of modules, or other factors. Routing and flow analysis algorithms are not a part of Multi-Batch, and are not discussed in this paper. The routing table settings in Multi-Batch are performed by routines in BKINIT when the background segment is loaded. These routines obtain routing data from BATCH.BAT, the initialization Zip-file.

Figure 2.1.3 shows the memory layout of IPC components in

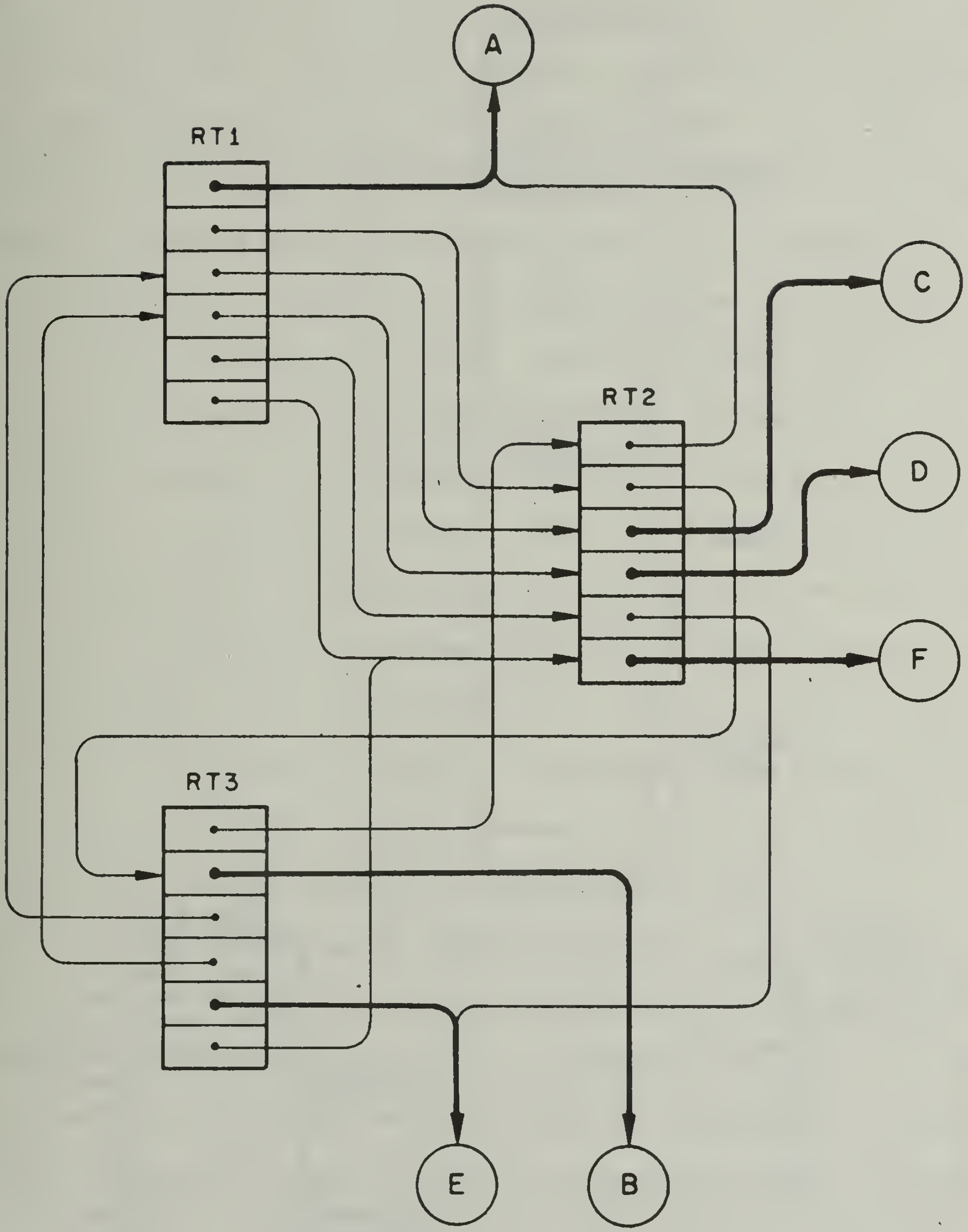


Figure 2.1.2

Routing Table Example

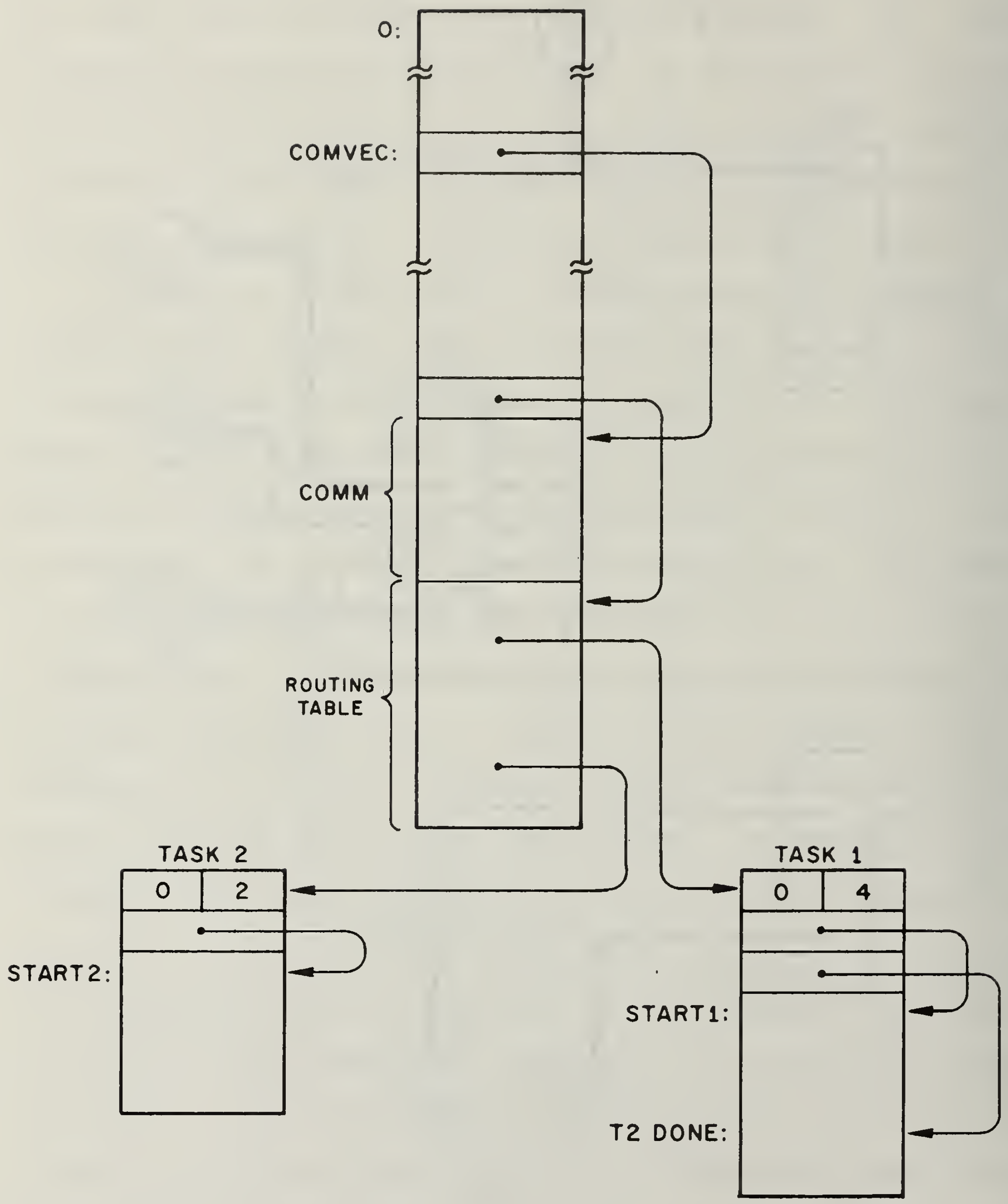
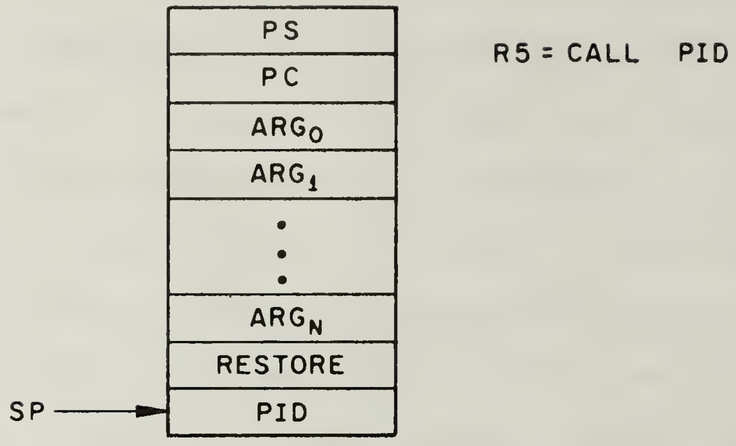


Figure 2.1.3

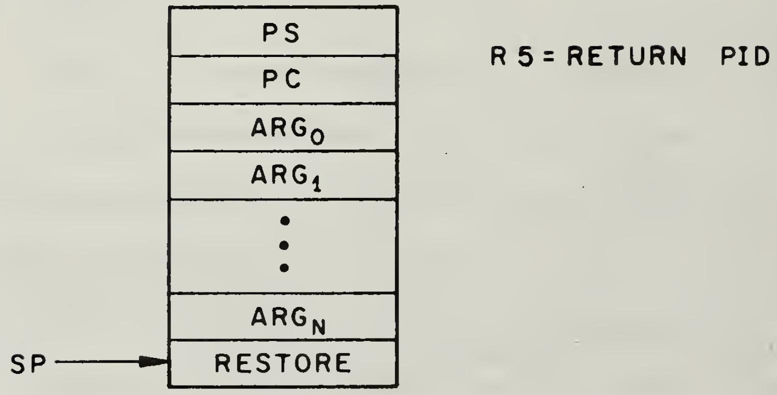
IPC Data Structure

Multi-Batch. Low memory adresses are at the top of the page. COMVEC is a word in low core unused by DOS. It is loaded by BKINIT with the address of COMM, the IPC traffic manager. The CALL macro generates code which locates COMM via COMVEC. This makes the calling sequence to COMM invariant with respect to changes in system size since the location of COMVEC is fixed. The memory word immediately preceeding COMM points to the start of the routing table. Thus the routing table can be located easily by foreground programs via COMVEC and the word preceeding COMM. Two entries in the routing table in figure 2.1.3 are shown pointing to in-core modules, TASK1 and TASK2. (TASK2 is shown with a single entry point, START2; TASK1 has START1 and T2DONE.) The first few words of each module are a prefix segment created by macro ENTRY. The first word is a control word. The other words form an entry table, containing the relative distances (indicated by arrows) to the entry points in each module. COMM accesses the entry table of a module when computing an interrupt address. The control word contains, in its lower byte, a count of the total number of bytes occupied by the entry table. The upper byte of the control word is reserved for use as a count of the number of bytes between the control word and the entry table in case the prefix segment format is expanded in the future.

In figure 2.1.4, SP denotes the stack pointer. PC and PS are the program counter and processor status. PC specifies the address of the instruction immediately following the CALL macro, i.e. the first return. PS specifies a setting of the processor status registers. In the current implementation, PS is



(a) CALL STACK FRAME



(b) ENTRY STACK FRAME

Figure 2.1.4

IPC CALL Conventions

arbitrarily set to zero since it is needed only for compatibility with hardware interrupt routines. PID denotes a 2-byte constant where the low-order byte is a module number assigned by NAMER, and the upper byte specifies an entry point within that module. Figure 2.1.4a reveals that the CALL macro first pushes PS and PC on the stack, followed by n arguments specified by the parameter list. RESTORE is a 2-byte constant where the lower byte gives the total number of bytes occupied by the arguments and the restore word itself. This count is used by the EXIT macro to pop everything off the stack except PS and PC. The upper byte of RESTORE gives the number of composite variables in the paramter list. By pushing all the composite variables on the stack, followed by the scalar variables, we can use the upper byte of RESTORE to efficiently drive the parameter copying routine from stack information. When the CALL macro transfers to COMM, hardware register five (R5) contains the PID (<m1,e1>) of the process being called. SP points to the top word on the stack which is the return PID (<m2,e2>) passed to the called process.

Figure 2.1.4b illustrates the stack frame passed to a called process. PS and PC are used by the EXIT macro to suspend from the communications interrupt. The other fields have the same interpretation as in 2.1.4a with the exception that the return PID is in R5. The PID of the calling process could also be supplied to the called process, although this is not done in the current implementation.

2.2 System Reliability

2.2.0 General Comments

Practical definitions of computer system reliability are usually related to the operating environment of the system. For example, military applications sometimes require that a system be capable of uninterrupted operation with no data loss in the event of localized hardware failure. On the other hand, a prototype system with a limited lifespan may need few reliability features. The objectives of Multi-Batch fall in between these two extremes since it is both an experimental and a production system. Our main requirement is that it provide continuous service for student batch processing. Since the system runs without an operator, the software must automatically recover from various situations that would normally cause an interruption of service. In practice this means that the system has three kinds of reliability measures: (1) preventive measures which ensure correct operation, (2) error handling procedures which deal with certain conditions as they arise, and (3) recovery procedures for situations not covered by (1) or (2).

Several preventive and error-handling measures were included in batch and used without change in Multi-Batch. These include Atkinson's interpreter and features added by Stocks and Kassel (see section 1.1). In the following paragraphs we describe the additions that were necessary to make Multi-Batch reliable.

2.2.1 Preventive Measures

The RESET instruction on the PDP-11 forces all I/O device controllers in the system to the idle state, cutting off I/O in progress. This instruction is used by several DOS routines, e.g. the KILL command for terminating jobs from the operator's console. It is obvious that RESET commands executed in the foreground part of Multi-Batch (i.e. by DOS) jeapordize the operation of the background segment. For this reason, RESET instructions in DOS were changed to calls on a new DOS service, the RESET routine. This routine keeps a list of devices not used by the background segment. When a RESET call is made this routine clears those devices in the list, thus avoiding any interference with the background segment.

Disk file space is limited on the system used for production with Multi-Batch. Therefore steps to conserve disk usage are necessary. Most of the disk overhead of Multi-Batch can be controlled. However, the structure of DOS prevents bounding the size of output files as they are created. It is possible for a job to generate large listing files, leaving little or no work space for succeeding jobs. The situation is complicated by the inability of the background segment to delete listing files after they have been printed. We approach the problem by limiting the number of entries allowed in the print queue. When the maximum number is reached, initiation of a new job is inhibited until a listing is completed. The software that implements this check also deletes finished listings. In a further attempt to reduce disk overhead, program CLEAN is called a number of times by most catalogued proc's. CLEAN deletes any finished print files. without CLEAN these files would remain on the disk until termination of the proc. Although our approach to the problem cannot guaranttee that over-allocation of the disk will never occur, it does limit the number of failures due to this problem.

2.2.2 Error-handling Measures

When a program terminates abnormally under DOS, it often happens that files used by the program are left open. That is, the DOS file system directory shows that the file is being used, even though that is not the case. When this happens, DOS prevents access to the file. Files in this "locked" state are most often encountered by the Multi-Batch routines that delete user files. A routine, DELREN, was provided which avoids these problems. To whit, DELREN is called by other routines that need to delete user files. DELREN attempts to delete a file; but if the file is locked, it unlocks the DOS file structure and then deletes the file.

2.2.3 Recovery Measures

It may happen that a program in the job stream produces no output file because of a user error. If another program, expecting the file, attempts to access it, DOS will return a "non-existent file" error. The programs that receive such errors are usually system programs used by Batch procedures, like the linkage editor or the assembler. These programs have no error recovery procedures of their own, and let DOS perform its own error processing. Error processing consists of typing a message on the console and suspending the system. Usually the only correct response to the message is a "kill" command. Students using the system often invent alternate recovery procedures which make matters worse. In order to eliminate such problems, we interrupt DOS's internal error process before it can print on the console and suspend the system. We then terminate the program that caused the error. We also terminate the current Zip-file by forcing BATCH.LDA to be the next program loaded. BATCH, which is the Multi-Batch monitor, prints a diagnostic message and proceeds to the next job. This sequence of events takes place without disturbing background operations to the card reader, line printer, or disk.

In the event of a total collapse of DOS, the system must be started from scratch. This means that the foreground job that was running when the system crashed is lost. But if the file system is still intact, Multi-Batch can "warm start" the line printer and card reader tasks using checkpoint data maintained in the file system.

3. DATA AND CONTROL FLOW

3.0 Conventions

Multi-Batch is organized into several modules, each of which contains one or more procedures. The next few subsections document data and control flows between modules and procedures in the system. Section 3.1 discusses the matter at the module level. Sections 3.2 and 3.3 examine the system at the procedure level. Section 3.4 examines the Zip-file interface between DOS and Multi-Batch.

The discussions that follow are keyed to a collection of directed graphs that depict relations between procedures and modules. Nodes in these graphs represent procedures or modules in the system, and are labled accordingly. In all graphs, except those in Section 3.1, procedure nodes appear on separate horizontal levels in the drawings unless more than one procedure from a module appears in the same drawing. In this case procedures from the same module appear on the same level in the drawing, and the associated module name is given in the left margin of the page. Figure 3.0.0 offers an example of the procedure labeling and placement conventions. Procedures from three modules are represented. Procedures A and B are each part of module 1; C and D are part of modules 2 and 3, respectively.

In the following, data flow graphs have a natural interpretation; that is, data follows the directed edges of the graph. The direction of an edge is indicated by an arrow-head drawn at one end of the edge. Figure 3.0.0 is again used as an example. Data flows along the arrows from INPUT to OUTPUT. The

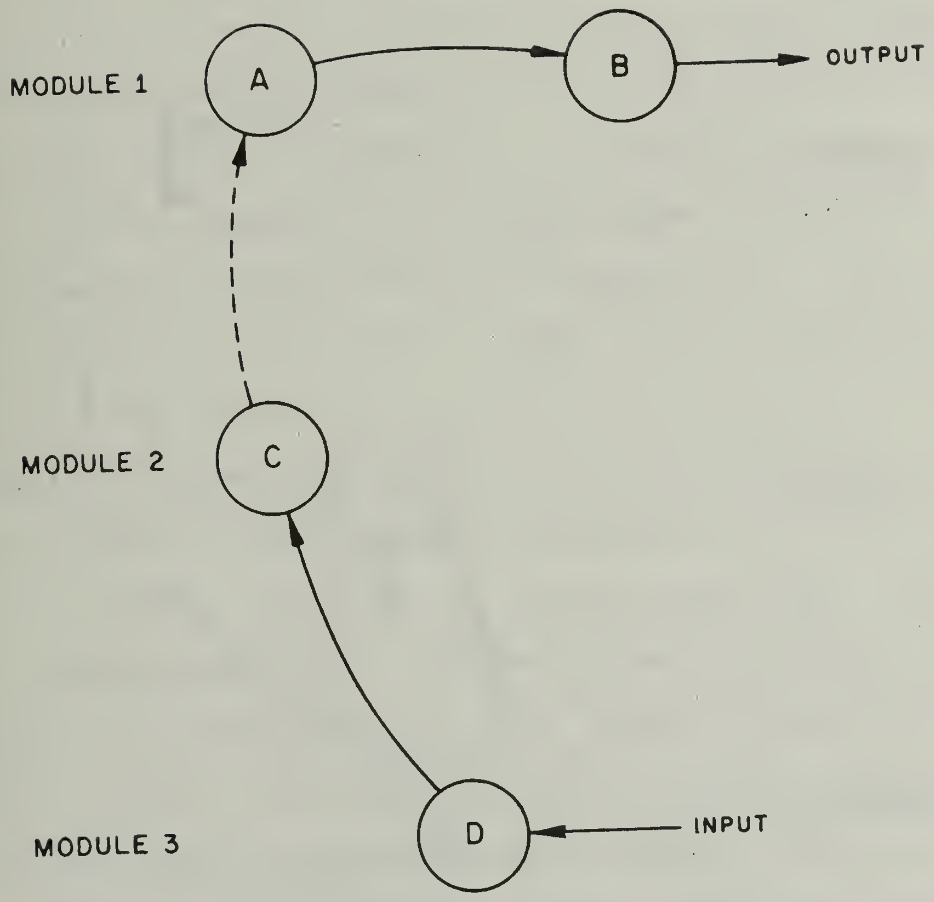


Figure 3.0.0

Prototype Data Flow Graph

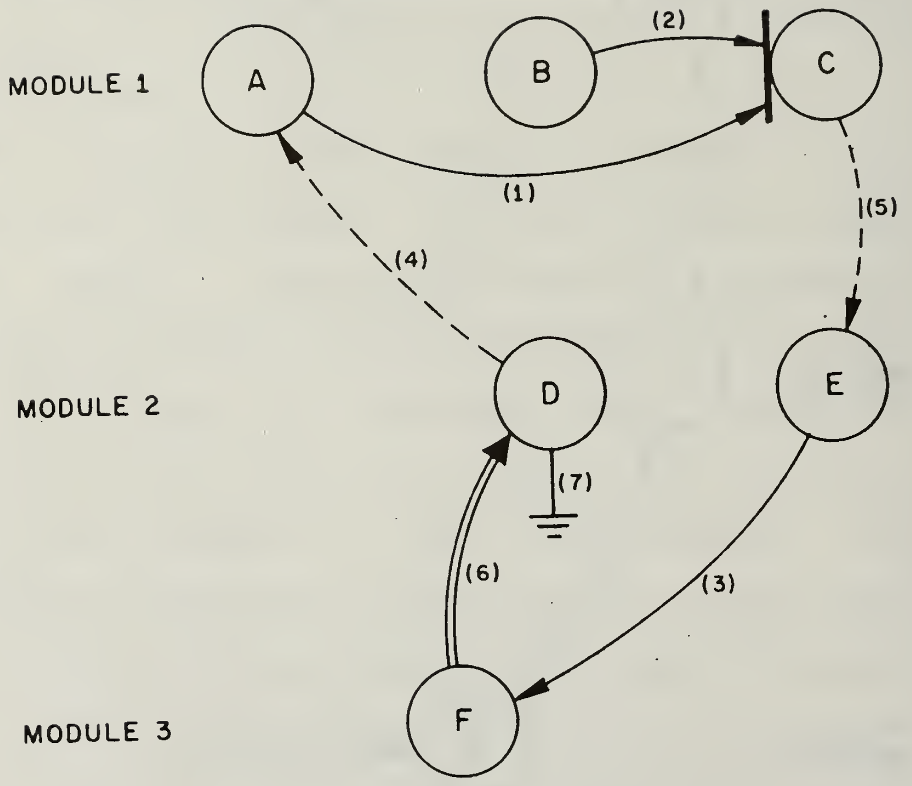


Figure 3.0.1

Prototype Control Flow Graph

dashed line arrow denotes data flow through the IPC software.

The arrow convention also applies to control graphs. However three different types of edge are distinguished in control graphs, each with its own special semantics. The three types of edge are indicated by solid lines, dashed lines, or double lines. Figure 3.0.1 provides an example of each. Solid lines (1,2,3) represent ordinary procedure calls. Dashed lines (4,5) denote IPC calls, and double lines (6) denote interrupts. The electrical ground symbol (7) is used to represent the exit from an interrupt routine. It was also found convenient to use a Petri net symbolism (the bold line tangent to C). The bold line means that C starts only after (1) and (2) have both been traversed in any order.

Given an ordinary procedure call, the return path from the called procedure is not diagrammed explicitly, but is presumed to follow the "calling" arrow in the reverse direction. Each solid edge in a control graph is traversed twice: once when a procedure is called (in the direction of the arrow), and once when a procedure returns from a call (in the opposite direction). In figure 3.0.1, control passes from A to C along (1), or from E to F along (3). Our convention is that C eventually returns to A (1), and F eventually returns to E (3).

Interrupts (6) are shown originating from the software module that controls the device. Each interrupt may cause control to propagate through several background routines until processing is complete. In figure 3.0.1, F is assumed to be the software control, or device driver, for some device. Assuming that transition (2) has been made, an interrupt (6) can cause the sequence (4,1,5,3,3,5,1,4,7). In this case transition (3) would signify a subsequent I/O request to driver F. Note that the interrupt exit symbol (7) may be attached either to the device driver or some other procedure, as shown in figure 3.0.1. What happens is that the interrupt call (6) occurs with only the interrupt exit information on the stack. This information remains on the stack when other procedures are called, so any procedure may call the DOS exit routine with this stack frame (7). One obvious restriction is that no IPC calls may occur between the interrupt (6) and interrupt exit (7), since procedures connected by an IPC call, e.g. A and D, may reside on different processors.

IPC calls (4,5) in Multi-Batch have two "returns." This is discussed in detail in section 2.1.3. The first return is made by the IPC traffic manager, COMM, to the instruction immediately following the CALL. This return is not diagrammed explicitly, and follows the reverse sense of the "calling" edge. The second return is actually another CALL through the IPC software, and is asynchronous with respect to the first return. That is, the second return may occur before, after, or during the first return. The second return is treated as a separate event in the system; indeed, it is often caused by a hardware interrupt. For this reason the second returns are shown on separate diagrams.

In some control graphs, a procedure is shown with several edges leaving it, indicating that the procedure invokes several other procedures. It is impractical to diagram the internal logic that governs the traversal of the control graph from such a node. Therefore, the text accompanying such graphs will explain the logical conditions that apply in each case.

3.1 Internal Organization of Multi-Batch

3.1.0 Control Hierarchy

The various programs, modules, and procedures that comprise Multi-Batch communicate either directly or indirectly. Direct communication implies the use of message areas in core or on the program stack. By indirect communication we mean information sharing via the disk file structure. Figure 3.1.0 shows all directly communicating modules in Multi-Batch with the exception of DOS interfaces to Multi-Batch which are described in Section 3.5. The module names shown in the diagram are the actual names used in the system. As such, they are abbreviations or acronyms derived from the functions that they perform. The module names shown in figure 3.1.0 are explained in table 3.1.0. Indirectly communicating parts of Multi-Batch are listed and explained in table 3.1.1.

Note that the connecting edges in figure 3.1.0 are labeled whth mnemonics. The mnemonics specify the type of software mechanism that connects the modules. "IPC" denotes a call through the IPC software using the CALL statement. "DDB" denotes a call on a DOS device driver equivalent to DOS's internal calls to device drivers. "EMT" denotes a standard call to DOS using the EMT instruction. "CO" denotes coroutine. And since the JSR instruction is normally used for procedure calls on the PDP-11, "JSR" denotes an ordinary procedure call.

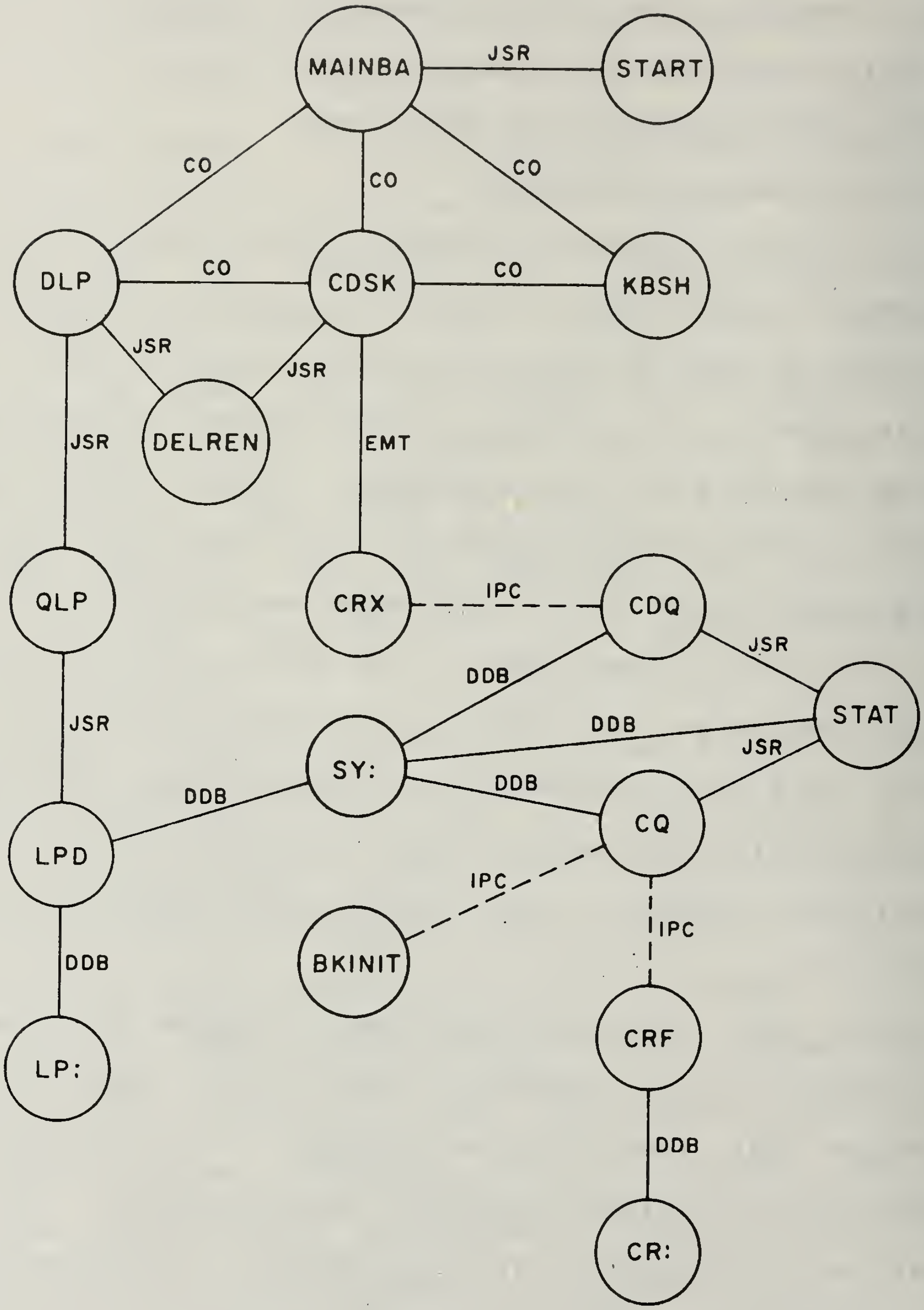


Figure 3.1.0

Multi-Batch Control Hierarchy

The coroutine convention in use here is as follows: MAINBA is a "parent" routine that invokes DLP, CDSK, and KBSH as three cooperating coroutines. The three coroutines "daisy-chain" continuously, i.e. control passes from DLP, to CDSK, to KBSH, and back to DLP again. Each coroutine continues to participate in the control loop until its processing task is completed. The processing tasks performed by the coroutines consist of simple data-copying operations which are described in the next section. Each coroutine terminates by transfering to an exit routine in MAINBA. This exit routine takes the place of each terminating coroutine in the daisy chain. When the exit routine notices that it is taking the place of all three coroutines, it issues a DOS exit call. At this point, output files for the previous job are being spooled, input files for the next job are ready, and KBT (the Zip-file processor) is primed to read the Zip portion of a catalogued proc. The DOS exit call terminates the current foreground process, MAINBA, and forces the DOS monitor to read from the console. Data for the next console read will be provided by KBT from the new proc, thus initiating the next BATCH job.

START	- executes the Zip file BATCH.BAT when MAINBA is entered the first time; does ABEND processing if MAINBA is entered via the DOS error processing sequence (see section 3.4).
MAINBA	- the "main program" part of Multi-Batch.
DLP	- the disk to line printer module taken from BATCH. It was modified to call QLP instead of the line printer.
CDSK	- the card to disk module taken from BATCH. It was modified to read from device CRX, an interface to background spooling, instead of the card reader.
KBSH	- the keyboard (console teletype) shell. KBSH handles teletype I/O for the job monitor. It also starts up the next proc when DLP and CDSK terminate.
DELREN	- deletes files from the disk. If a file is "locked" it unlocks the DOS file directory, then deletes the file.
QLP	- the interface between DLP and the background line printer daemon (LPD). QLP "queues the line printer" by adding a file to the queue of output files.
LPD	- the line printer daemon. LPD copies text from disk files to the line printer.
LP:	- the line printer driver.
SY:	- the disk driver.
CRX	- the "virtual" card reader. CRX interfaces Multi-Batch to the background software, reading the next job from the input queue.
CDQ	- the card input "dequeuer" which obtains records from the input queue.
STAT	- maintains a job queue in core and on disk. The job queue contains, among other things, the physical disk address of the first record of each job.
CQ	- the card input "queuer" which attempts to keep the disk queue filled with input card images.
CRF	- the card reader formatter. It calls the card reader driver, and formats card images into 510 byte records that are passed to CQ.
CR:	- the card reader driver.
BKINIT	- the background initiator.

Table 3.1.0
Multi-Batch Modules

CON	- the configuration program. It initializes Multi-Batch's internal communication files: STAT.BAT, STATUS.COM, and SPOOLI.CRD.
CLEAN	- uses a copy of DELREN (see table 3.1.0) to delete files. The names of files to be deleted are entered from the console. CLEAN is called from strategic points in catalogued proc's for file cleanup operations. When the input list of file names is exhausted, CLEAN deletes any output spooling files (see section 3.3) that have been printed.
MAKER	- a program that builds catalogued procs on the disk from card input.
S	- a program which includes BKINIT and most of the background segment. S uses configuration data located in STATUS.COM.

Table 3.1.1
Indirectly Communicating Modules

3.1.1 Data Flow

A large portion of Multi-Batch is dedicated to transporting user programs and data through the system. In the foreground part of the Multi-Batch monitor these data transfers are file-copying operations. Figure 3.1.1 illustrates these file transfers. User input is copied to DOS files having names of the form GO.EXT, where EXT is an arbitrary three-letter file extension. Output files created during the execution of a proc, destined for the printer, adhere to the same GO.EXT naming convention. The output files are concatenated together by DLP into a single file, SYSOUT.OUT, which is then passed to QLP. QLP places SYSOUT.OUT at the end of a queue of files waiting to be printed, and in doing so changes the name to SYSOUT.AAA, SYSOUT.AAB, etc. STAT.BAT and STATUS.COM are system control files. They are read by DLP and updated by DLP and KBSH. Except for STAT.BAT, all other files in the system with extension BAT are catalogued procs. MACRO.BAT, for example, is a proc on the system that assembles a user source program, loads and executes it under the interpreter.

Starting at the top of figure 3.1.1, user input is shown entering the system via CDSK. If background input spooling is disabled, CDSK obtains user input directly from the card reader. When background spooling is in effect, the background software controls the card reader, and CDSK obtains input from the "virtual" card reader instead.

The card deck format expected by CDSK is indicated by the label, USER INPUT. Control cards are distinguished from other

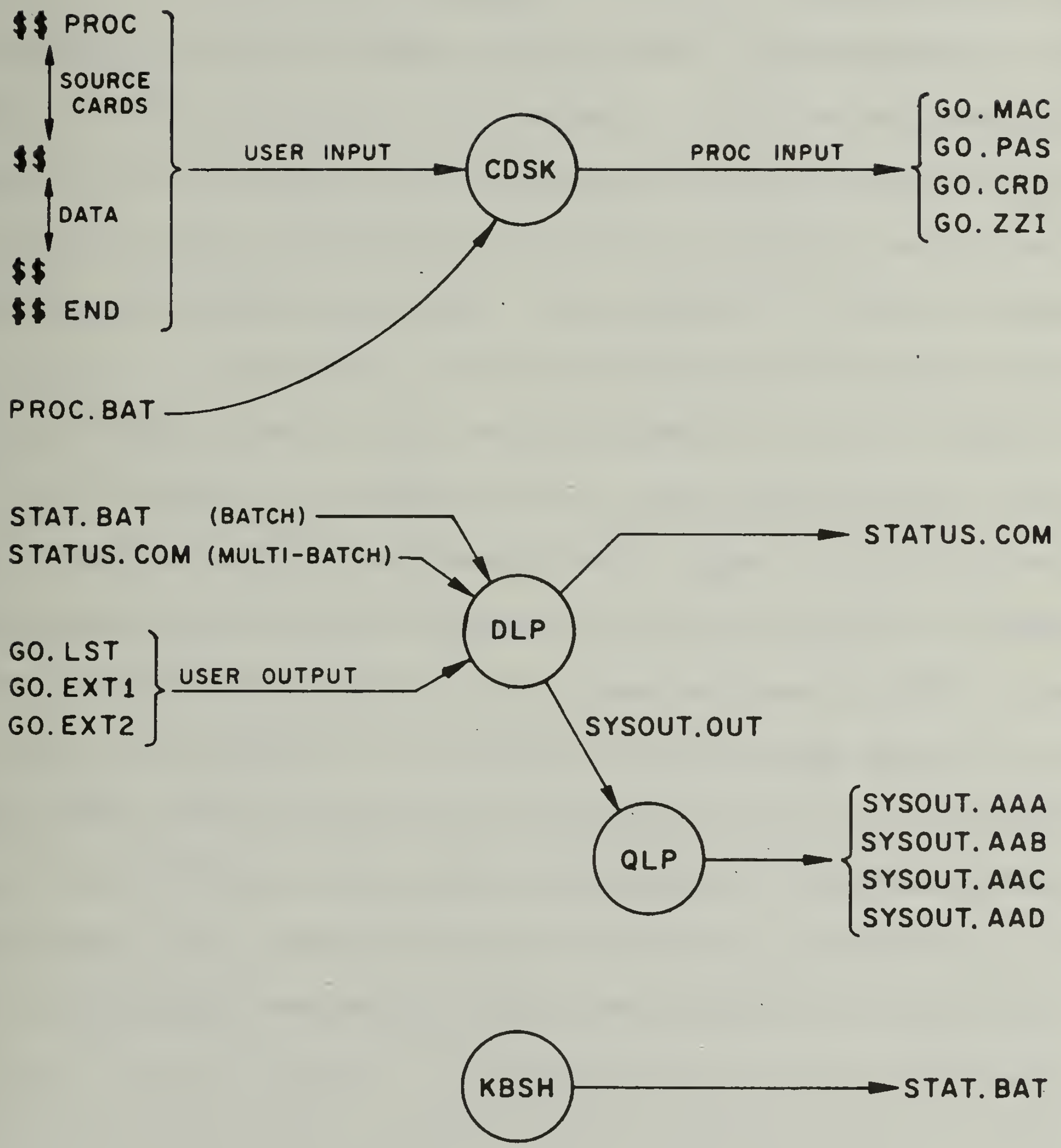


Figure 3.1.1

Data Flow in Multi-Batch

other cards by the presence of a dollar-sign in card columns one and two. The control cards are called $$ cards. The first $$ card in a typical card deck specifies the name of a catalogued proc that the user wishes to invoke. For example, if the first card were $$MACRO, the system would attempt to invoke a proc catalogued as MACRO.BAT. Cards with $$ in the first two columns and blanks elsewhere are recognized as end-of-file markers by the system. They divide the continuous stream of input records into logical files. $$END cards are recognized as end-of-deck markers, and are not copied into the job queue.

CDSK reads the input stream until a valid $$PROC record is found. The PROC.BAT file will contain a list of file name extentions for input files needed by the Zip part of the proc, as well as a similar list of output files to be generated by the proc. CDSK reads logical files from the input stream until the input list of extentions is satisfied. The list of output file extentions is copied onto the stat file STAT.BAT. When the proc terminates, MAINBA is once again entered, and the list of output file extentions is retrieved from STAT.BAT. DLP uses this list to locate the output files that will become SYSOUT.OUT.

3.2 Input Spooling

3.2.0 Data Flow

The background input spooling software manages a fixed-size area of contiguous disk space on the system disk. The disk space is actually a DOS file known to the system as SPOOLI.CRD. The size of this spooling area is a multiple of 16 disk blocks, where each block contains 510 bytes of data and a 1 word link to the next block. The exact number of multiples of 16 is given as a parameter to program CON when Multi-Batch is initially executed. The spooling software can be thought of as a producer-consumer system. That is, one part of input spooling, CQ, is a "producer" because it attempts to keep SPOOLI.CRD completely filled with card reader input. The "consumer" part of the system, CDQ, drains records from SPOOLI.CRD for CDSK. CQ and CDQ are synchronized by procedures and common data structures located in module STAT. The file, SPOOLI.CRD, is referred to as the input queue because it is organized as a queue of files. The data flow in and out of the input queue is diagrammed in figure 3.2.0. The card reader is driven from module CRF, the card reader formatter. CRF packs input card images into 510-byte buffers. CRF also performs some scanning functions on $$ cards. The dotted line between CRF and CQ in figure 3.2.0 indicates that these two modules communicate via the IPC software and may therefore reside on separate processors. The same is true for CDQ, CRX1, and CRX2. The IPC connections between the CRX's and CDQ allow an input queue to reside on one processor equipped with a large disk system. Then copies of Multi-Batch on other processors may be

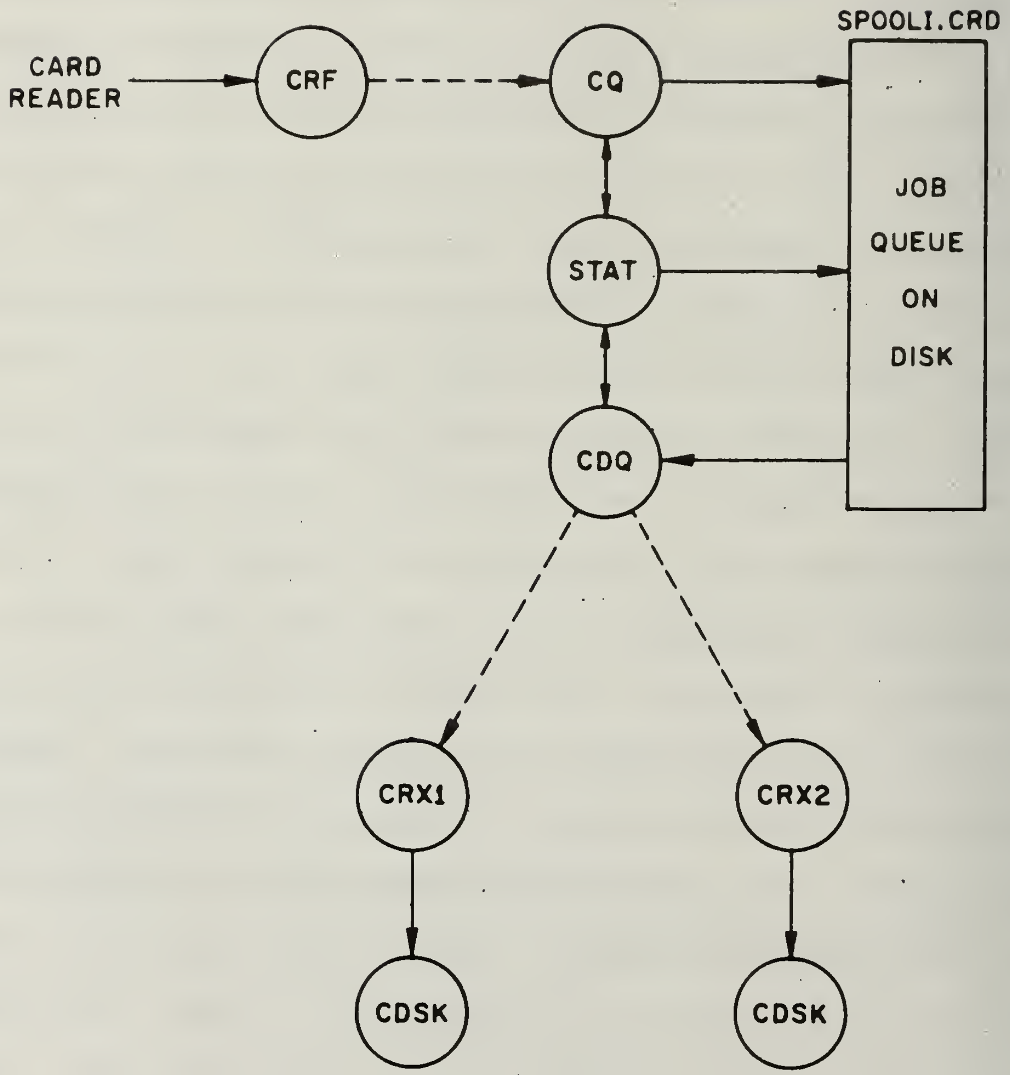


Figure 3.2.0

Input Spooling Data Flow

"driven" from the centralized input queue. Other arrows in the drawing indicate communication between STAT and the data-transfer modules, CQ and CDQ, as well as the STAT process that updates the background status page in SPOOLI.CRD. The status page is updated from an in-memory page after every complete file transfer in or out of the queue. This permits system checkpoint/restart procedures. In the current implementation, a system restart is treated as a cold start; i.e. the status page is overwritten with an empty one. This is because the code necessary to check the validity of information in the old status page has not yet been implemented.

3.2.1 Spooling Initialization

Figure 3.2.1 depicts the initial system event. BKINIT is the loader program that inserts the background segment into DOS's resident monitor area. After loading the background segment, BKINIT calls CQ through the IPC at entry point SYSINIT. This is indicated by (1). At this point the input queue will be empty of jobs. Therefore, when SYSINIT calls procedure ALLC (2) in STAT to allocate a disk block in the input queue, the call will succeed. In other instances, calls to ALLC may not succeed, and ALLC will return to the caller. However, when a call does succeed, as it will during system initialization, ALLC calls BLKFREE (3) in CQ. BLKFREE, in turn, starts up the card reader via CRF (4,5). The reason that ALLC calls BLKFREE instead of returning a boolean value to the caller is that ALLC is called from more than one location in the background software. It is simpler to let ALLC start or restart the card reader process than

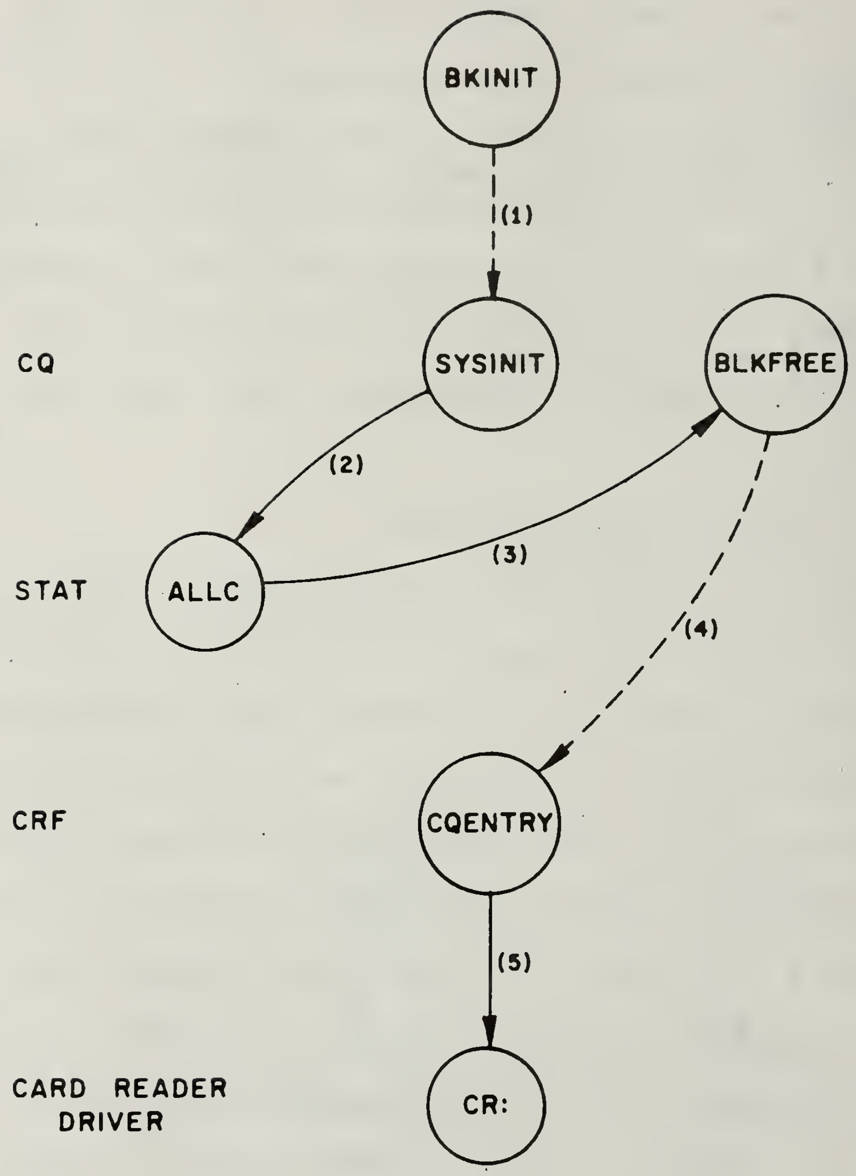


Figure 3.2.1

Input Spooling Initialization

to duplicate calls to BLKFREE at several places in the system. Once the card reader has been started, the procedures invoked in figure 3.2.1 return to BKINIT.

3.2.2 Input Queue Operation

The card reader operates "in parallel" with the rest of Multi-Batch until a complete card has been read. The card reader interrupts the card reader driver once for every column on the card. These interrupts are not considered here because they are "invisible" to the rest of the system. However, on the last (80th) such interrupt the card reader driver returns to CRDUN in CRF. This is indicated by (1) in figure 3.2.2. A control transition like (1), caused by the completion of an I/O task, is called a completion return. Since CRF must read several cards in order to fill its 510-byte buffer, CRDUN may either call CR: for another card (2), or pass a completed buffer to CRFDUN (3). Note that the card reader driver trims trailing blanks from the right-hand sides of card images, and CRF merely concatenates these card images until the 510-byte character count is satisfied. The last card in the 510-byte buffer may extend past the main buffer into a small overflow buffer. When the next read request is received by CRF, the overflow buffer is copied to the front of the main buffer. Another special case arises with the $$ cards. CRF checks each incoming card for $$ in the first two positions. When a $$ card is found, CRF is required to place the control card in its own 510-byte buffer. Therefore, receipt of a $$ card terminates the buffer-filling operation, and the $$ record is placed in the next buffer. The buffer overflow

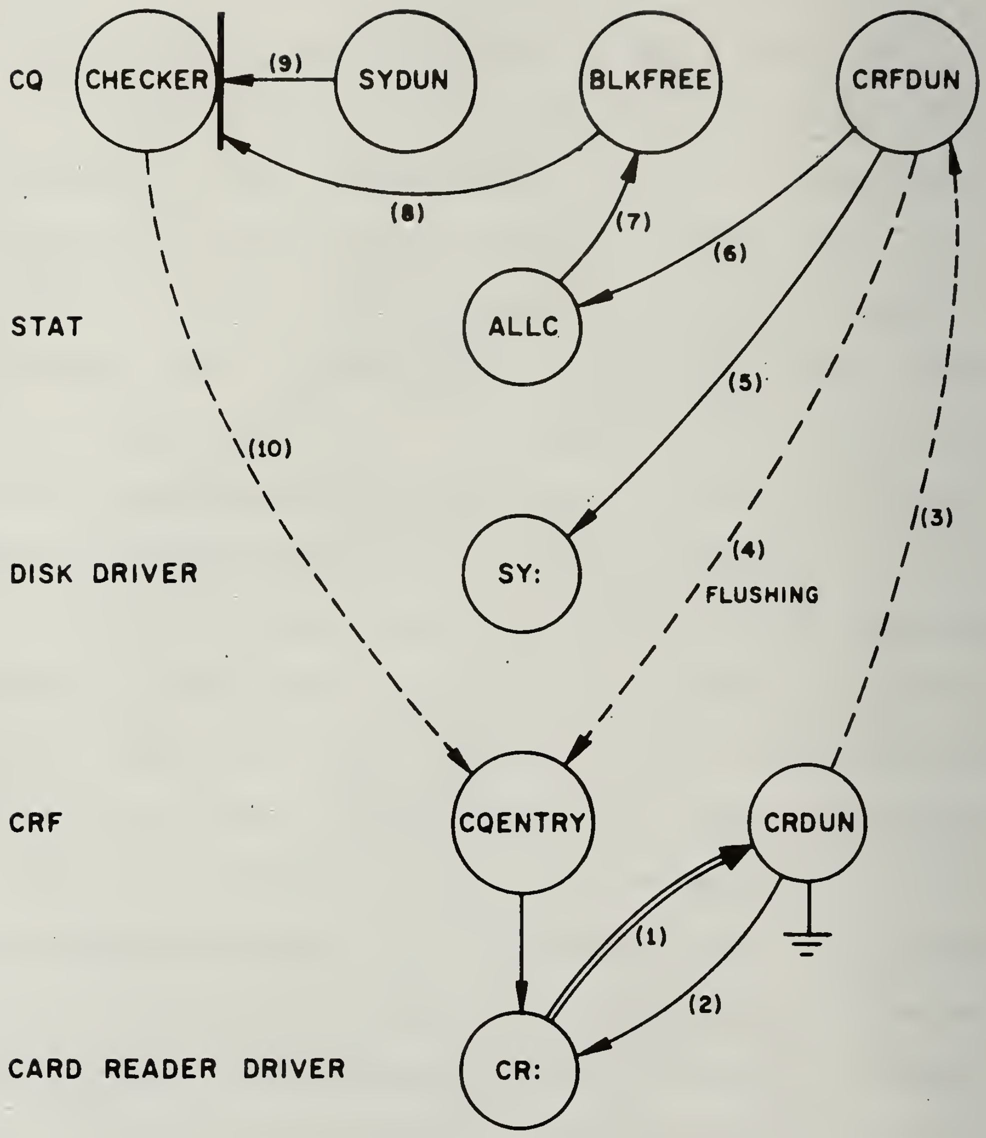


Figure 3.2.2

Card Reader Interrupt: Text

operation and $$ card processing are both invisible to CQ and are not shown in control graphs given here.

A disk buffer received by CRFDUN is either a $$ card buffer or a text buffer. The text case is illustrated in figure 3.2.2, and the $$ card case is shown in figure 3.2.3. Since CQ "knows" that every job must begin with a $$PROC card, all data from CRF is discarded until a $$PROC card is found. The data-flushing operation is accomplished by calling CRF for more data, and then exiting (see (4), figure 3.2.2). In the event that a "valid" text buffer is received by CRFDUN, the data is first copied to the input queue (5), then ALLC is called (6) to obtain a disk block for the next input buffer. If ALLC fails, then nothing else happens, the called procedures return, and the input process goes to sleep until a block becomes free via the dequeuing process (see (3), figure 3.2.7). If ALLC succeeds, then transitions (7) and (8) are taken. Procedure CHECKER determines whether or not to request another buffer from CRF. This decision depends on whether the disk transfer started by (5) has finished or not. This step is necessary because CQ is single-buffered due to the limited availability of memory for the background segment. Therefore, the buffer in CQ must be copied to the input queue before it can be loaded again from CRF.

The preceeding paragraphs outline the control procedures involved with copying text from the card reader to the input queue. The control sequence for processing $$ cards consists of adding a procedure call to STAT just prior to transition (5) in figure 3.2.2. This extra call is shown in figure 3.2.3. The

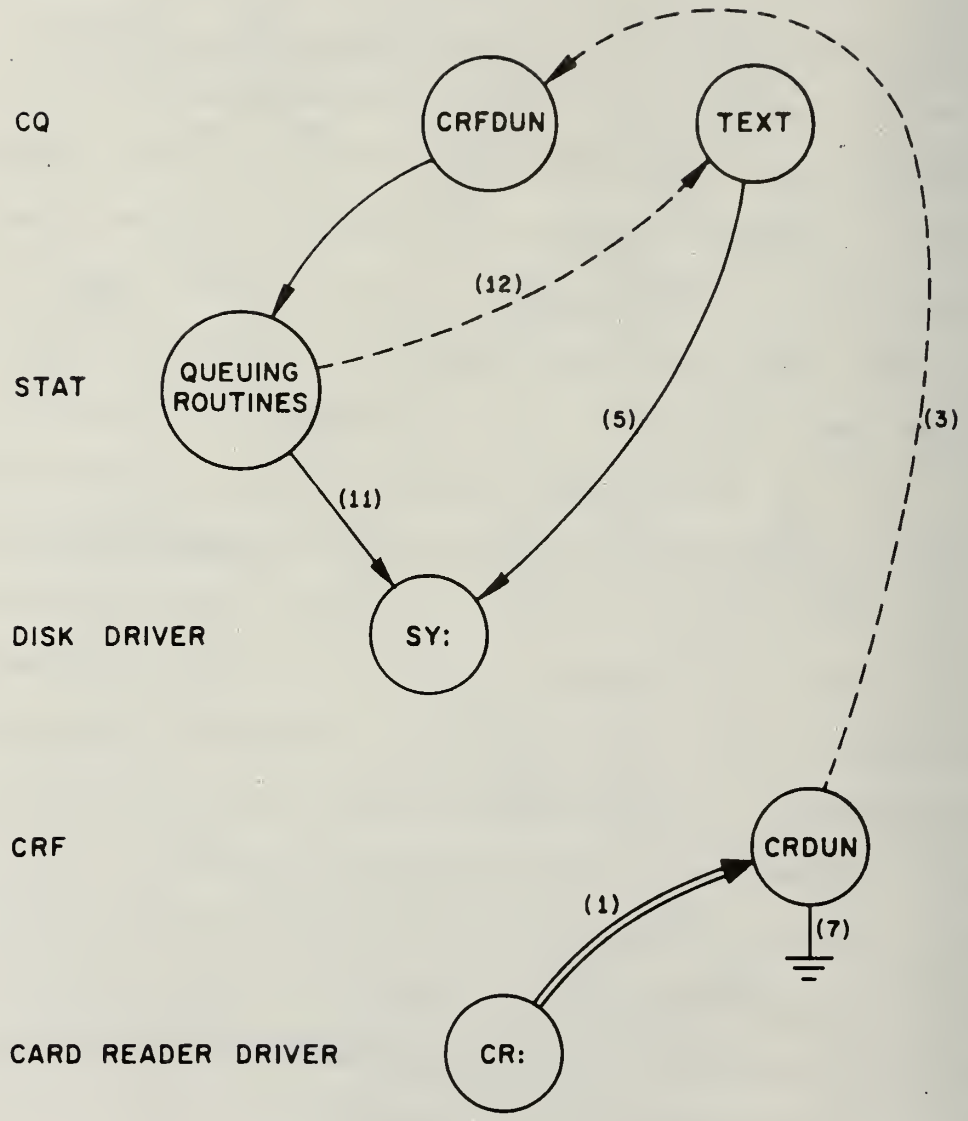


Figure 3.2.3

Card Reader Interrupt: $$Card

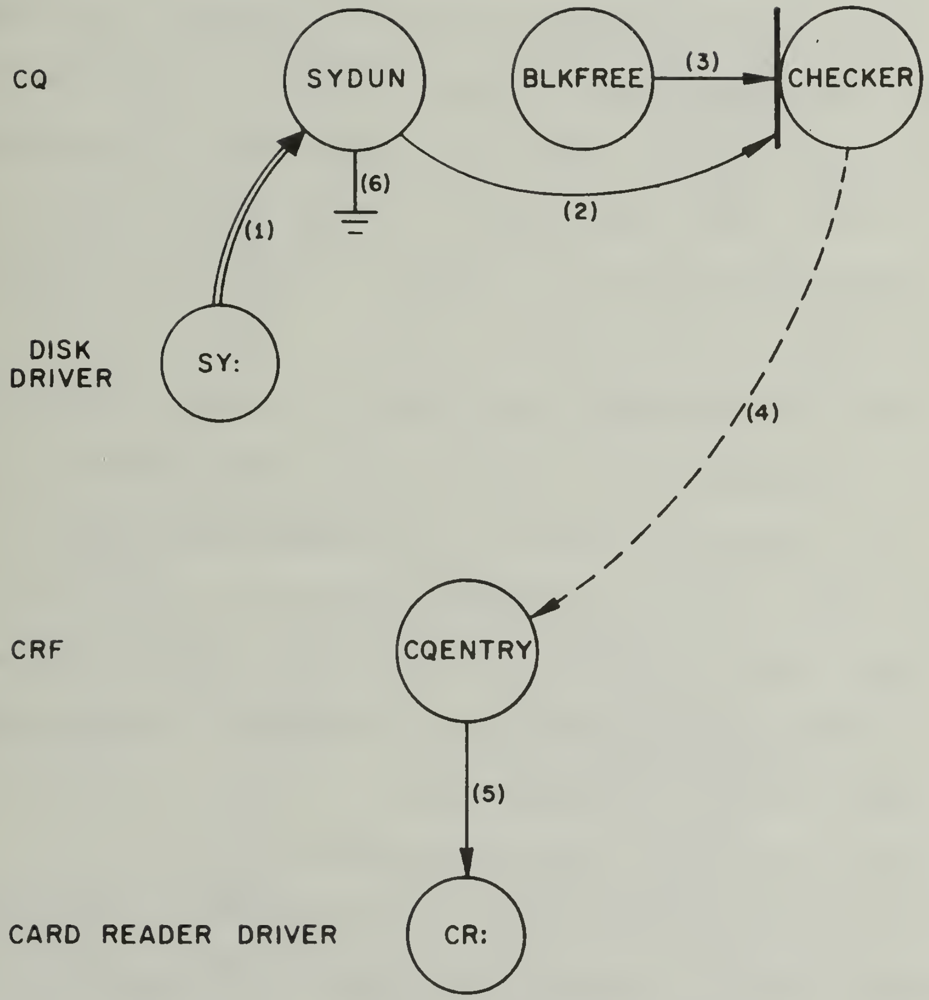


Figure 3.2.4

Disk Interrupt: Write

numbered transitions in figure 3.2.3 correspond exactly to similarly numbered transitions in figure 3.2.2. When a $$ card buffer arrives at CRFDUN (figure 3.2.3) it may mark the beginning of a new job and the end of the previous one, or it may only indicate the end of a logical file. In any case, CRFDUN will call appropriate procedures in STAT to update the status page. The new status is saved in SPOOLI.CRD (11), and then an IPC call is made to CQ at TEXT (12). The TEXT entry in CQ writes the $$ card to the input queue as if it were a text buffer. Execution then continues as shown on figure 3.2.2 with transitions (5) through (10).

Except for the case where input records are flushed, the control sequences in figures 3.2.2 and 3.2.3 initiate a disk write, and then go to sleep. The wakeup caused by the completion of the disk write is shown in figure 3.2.4. The disk driver interrupts CQ at SYDUN (1). SYDUN calls CHECKER (2), and another disk buffer may be requested from CRF (4) if there exists a free disk block in the input queue (3).

3.2.3 Card Reader Error Processing

Card reader device errors cause a special interrupt which is processed in CRF at entry point CRERR. Although error interrupts can be generated by electronic malfunctions in the card reader, they usually appear as a result of turning the card reader off, or by running out of cards in the input hopper. All error interrupts from the card reader are processed using the same algorithm. That is, we observe the card reader status register at 1 second intervals until the error clears, then restart the

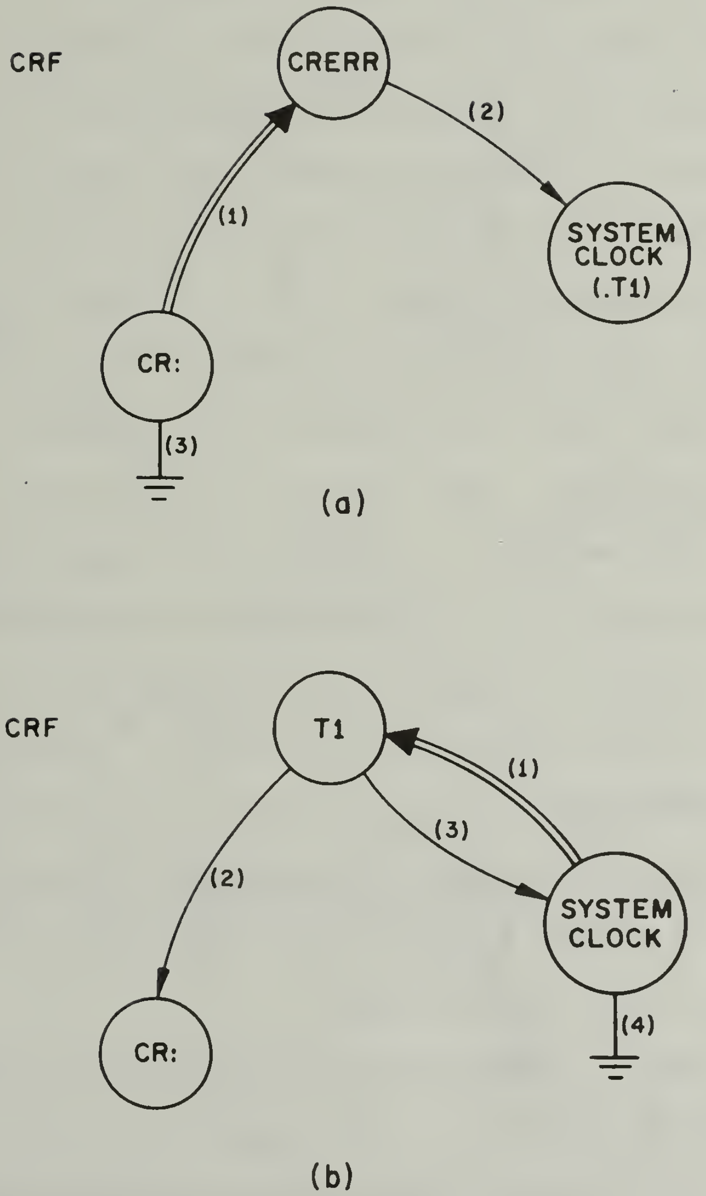


Figure 3.2.5

Card Reader Error Handling

reader. Figure 3.2.5 indicates the modules that are involved. In figure 3.2.5a, CRF calls an interval timer (2) when the error interrupt is generated (1), and then returns to the card reader which exits from the interrupt (3). Figure 3.2.5b depicts the waiting process. The system clock wakes up CRF at T1 (1) after a 1 second delay. If the card reader still shows an error, T1 sets another time-out (3), and exits back to the clock routine (1,4). Otherwise T1 restarts the device (2), and exits back to the clock routine (1,4) without requesting another time-out.

Note that ".T1" is the name of a counter variable in the system clock routine. This variable is associated with interrupt entry T1. One requests a time-out interrupt at T1 by setting a non-zero value into .T1. This value will be interpreted as the number of 1/60th of a second clock ticks that the clock routine should count before interrupting at T1. A similar relationship holds between label T2 and variable .T2 in the clock routine. The current implementation has four timer variables: .T1 through .T4. Two of these are used by Multi-Batch, and two are free.

3.2.4 DOS Input Queue Access

Module CRX is the interface between the DOS I/O subsystem and the background segment. CRX appears to DOS as a normal device driver. However, DOS read requests to CRX are translated into read requests from the input queue. The control steps that comprise a data-transfer from the input queue to CDSK are shown in figures 3.2.6 and 3.2.7. Figure 3.2.6 follows a read request up to the point of initiating a disk transfer. Figure 3.2.7 traces the activity accompanying the completion of the disk

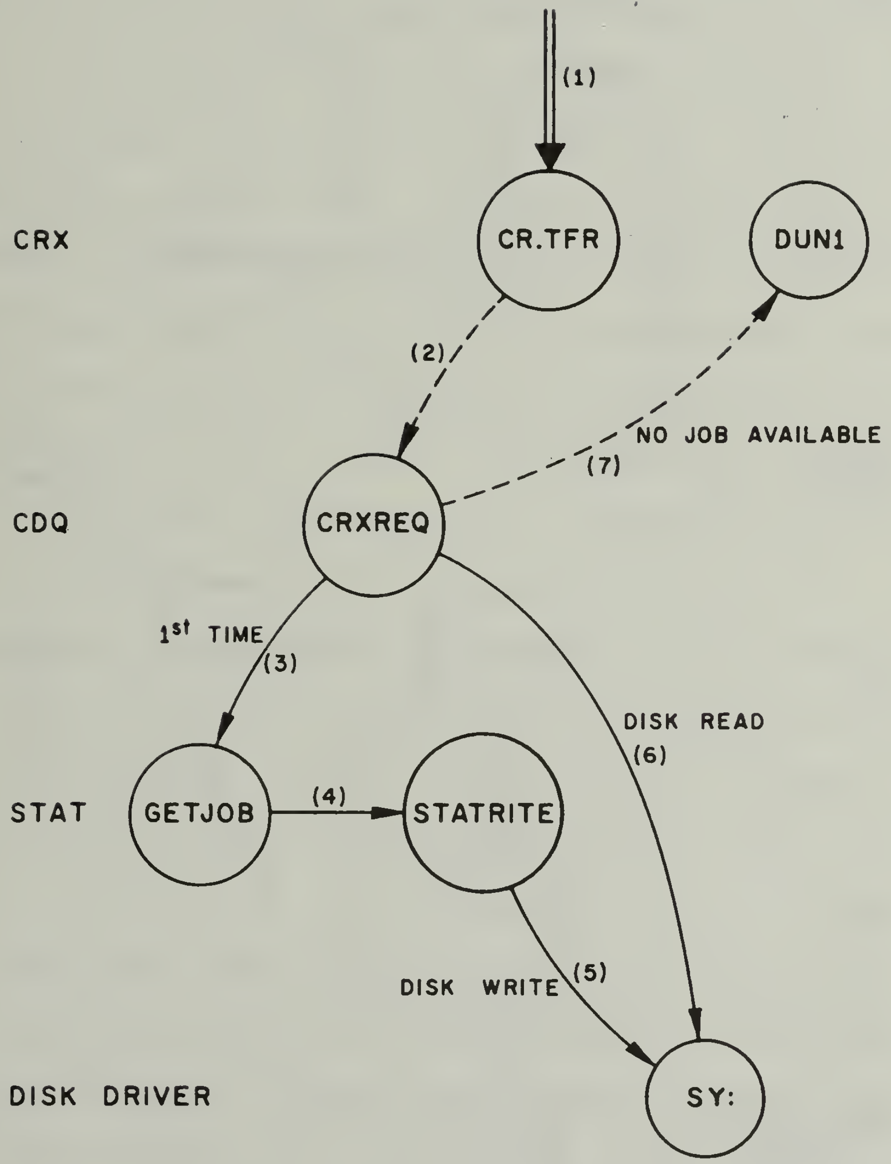


Figure 3.2.6

Input Read Request

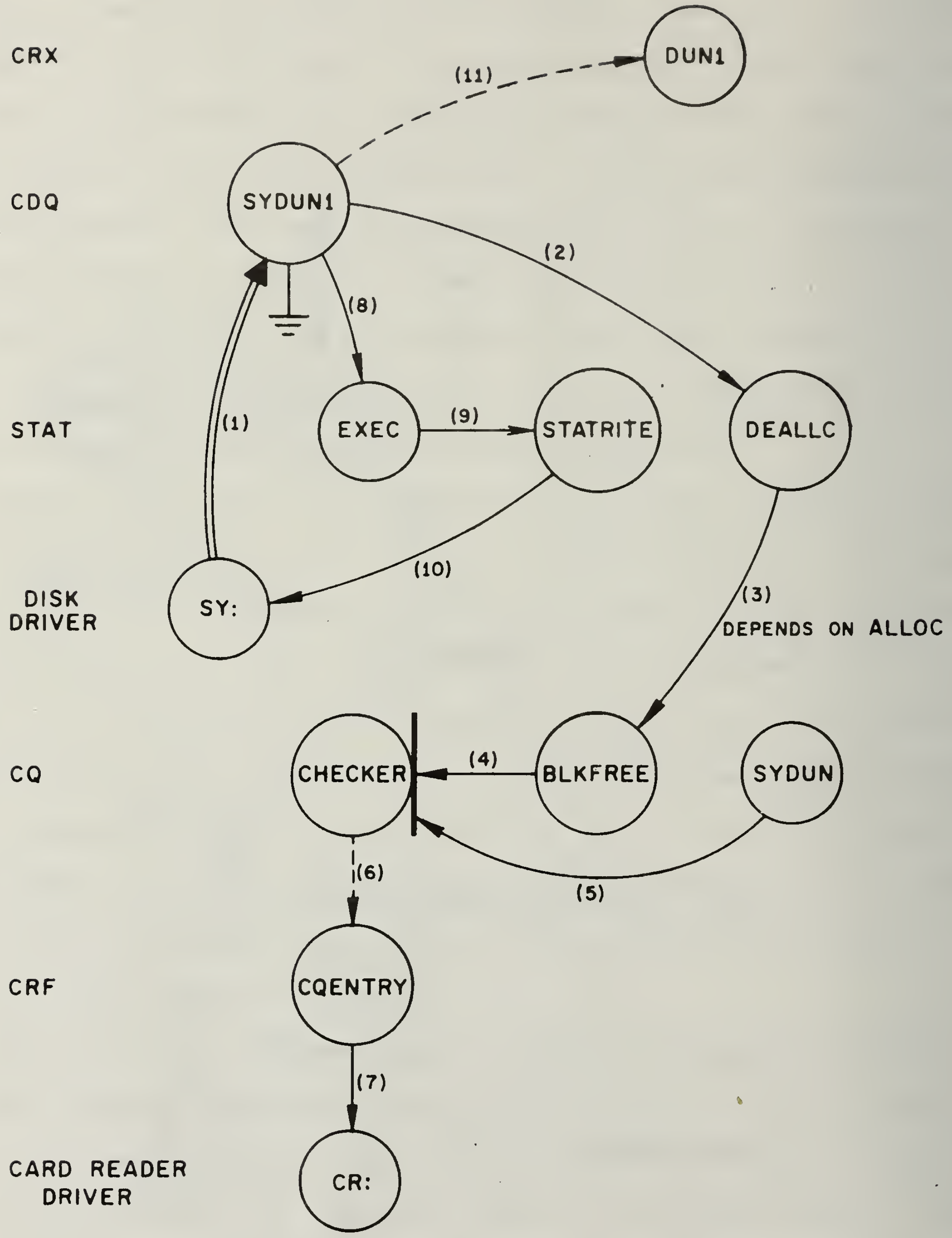


Figure 3.2.7

Disk Interrupt: Read

transfer.

In figure 3.2.6, the DOS read request (1) is interpreted at CR.TFR within CRX. CRX calls CDQ (2) through the IPC. Initially, CDQ must find a job in the input queue. This is done by calling STAT (3). If the input queue is empty, GETJOB returns immediately to CDQ with a "queue empty" value, at which point CDQ takes a completion return to the driver (7). CRX at DUN1 will set an error flag for the DOS I/O system, indicating a read failure. CRX also emulates the error-trapping function of the normal card reader driver. If the error-trapping option is enabled and DUN1 is entered with a "no job available" code from CDQ, then the special error return is taken. In Multi-Batch, CDSK calls CRX again whenever this error trap occurs. The process of calling and trapping continues until a job becomes available. When this happens, transitions (4) and (5) in figure 3.2.6 are taken in order to update the status page in SPOOLI.CRD. Then the first block of the job is read (6). Exits are made through (1), and the background software sleeps until the disk interrupt from (6) occurs.

When a disk read from SPOOLI.CRD is complete, the disk driver interrupts CDQ (see (1) in figure 3.2.7). The first order of business is to return the newly arrived disk block to the free list of input blocks (2). This step always causes a common variable in STAT, called ALLOC, to be checked. ALLOC records whether or not CQ is waiting for a free block. If CQ is indeed waiting, BLKFREE is called. This may result in a call to CRF according to the conventions previous described for figures

3.2.4 and 3.2.2. After the synchronization with CQ has been done, the just-read disk block is passed to CRX (11) for return to CDSK. However, if the block is also the last block of a job, the EXEC procedure in STAT is called before invoking the DUN1 completion call. EXEC removes all traces of the just-read job from the input queue.

3.3 Output Spooling

3.3.0 Data Flow

Output spooling in Multi-Batch consists of the following: generation of listing files, management of an output queue of listings, and copying of listing files to an output device. Listing are produced by module DLP, as described in section 3.1. Copying operations are performed by the line printer daemon (LPD); and the output queue is managed jointly by LPD, QLP, BKINIT, and program CLEAN.

3.3.1 Output List

The output list The output list resides in the background is a common data structure accessed by output queue management routines. It resides in the background segment as part of the line printer daemon. A copy of the output list is maintained in file STATUS.COM for use by system restart and recovery procedures. The output list, illustrated in figure 3.3.0, is located in the background segment below DOS's bitmaps. DOS's BFS pointer always points to the first bitmap word. Since the BFS pointer can always be located by foreground software, and because the size of RMON and the background segment is subject to change, all access to the output list from foreground software is

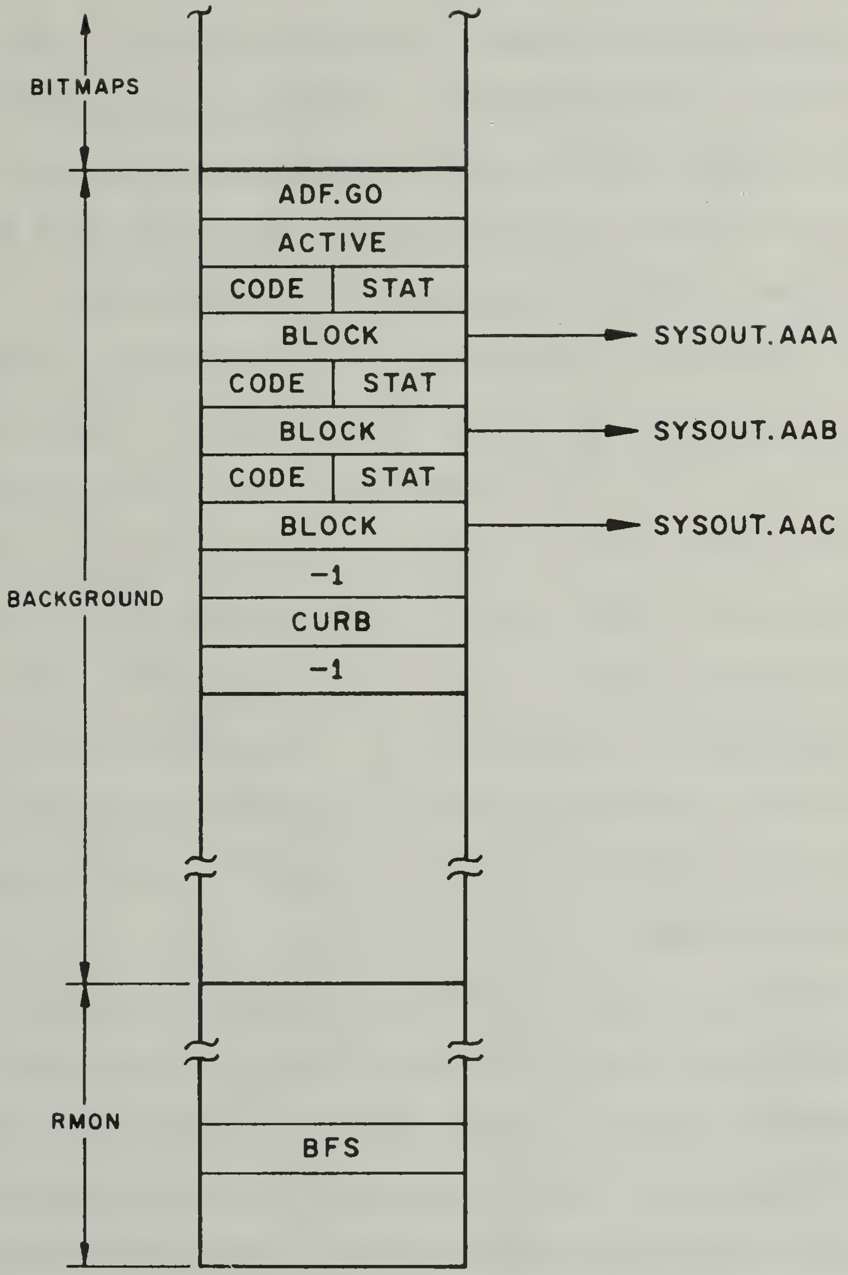


Figure 3.3.0

Output List

accomplished via BFS. The first word below the bitmaps (ADF.GO) contains the physical address of the initiation entry point of LPD. The second word (ACTIVE) indicates whether or not LPD is actively spooling data. ACTIVE is zero if LPD is idle; otherwise, it contains the address of the currently active BLOCK entry. A BLOCK entry contains the starting disk block number of a listing file. Associated with each one-word BLOCK entry are a STAT byte, a CODE byte, and a unique file name. The last entry to be mentioned is CURB, which is the current block number. CURB is always updated with the block number of the most recently read disk block when spooling operations from disk are in progress. The minus one's in the output list are markers used as terminator flags for scanning or copying the output list. All software that accesses the output list starts at the "top" and proceeds downward until a minus one is encountered. This convention permits the output list size, and hence the output queue size, to be adjusted (by moving the minus one's) without re-assembling all of the software.

Figure 3.3.0 shows the output list configured for a maximum queue size of three listings. The file name associated with the first BLOCK entry is SYSOUT.AAA. If SYSOUT.AAA does not exist in the system at a given time, the BLOCK entry will be zero. If the file does exist, the STAT byte will indicate the current state of the file. STAT will be set to one if the file is waiting to be printed, minus one if the file is being printed, or zero if printing is complete. CODE is used to indicate a source and destination for data transfers. The lower four bits of CODE can specify up to 16 sources for input to LPD; the upper four bits can specify equally many destinations. By convention, both the system disk and line printer are represented by zero codes. Thus, for a single-processor system with one disk and one line printer, CODE will always be zero. When more devices are in use, the four-bit source and destination codes in CODE can be used to specify minor device numbers to a device driver, e.g. a second line printer. In a multiple-processor system, certain source and destination codes would specify that transfers be made through the IPC software. These codes would be logical process identifiers for use in a CALL statement.

3.3.2 Two-processor Example

Figure 3.3.1 indicates the data flow for a two-processor system with a single line printer. Each DLP passes the starting block of an output file to QLP. As explained in section 3.1, output files from DLP are always named SYSOUT.OUT. QLP searches the output list for an entry with a zero in the STAT field, indicating that the file associated with that entry has finished printing. If no entries are available, the the queue is filled. In this case, QLP prints the message "OUTPUT QUEUE FULL," and waits until LPD finishes copying a file. When a free list position is found, QLP may have to delete a recently printed SYSOUT file before proceeding to the next step. If BLOCK is non-zero, the associated SYSOUT file is deleted; otherwise, the file does not exist, and no action is taken. At this point SYSOUT.OUT is renamed to SYSOUT.AAA, SYSOUT.AAB, or SYSOUT.AAC depending upon which BLOCK entry was selected in the previous

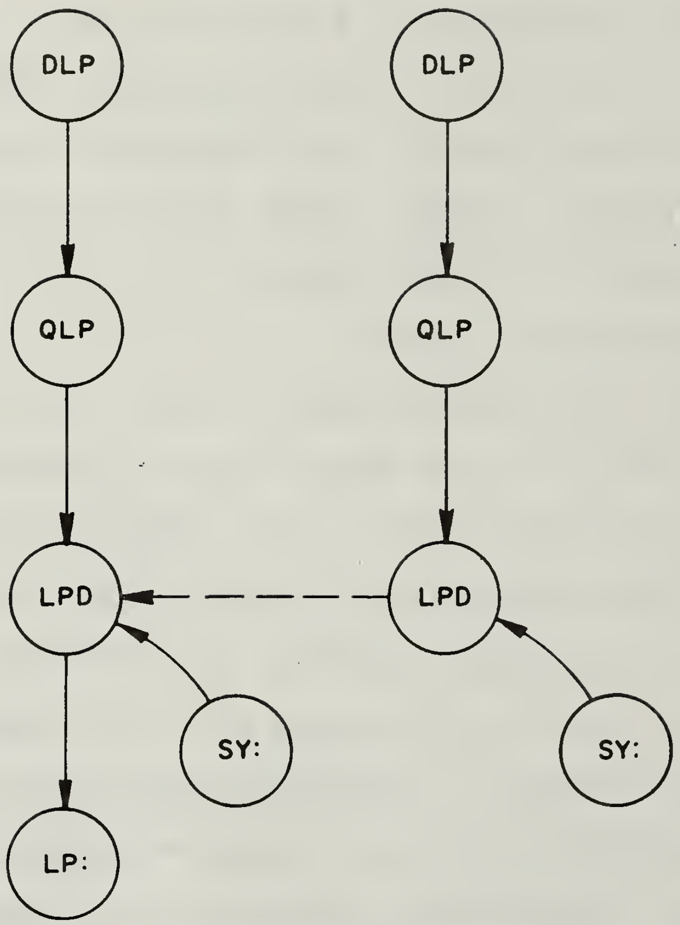


Figure 3.3.1

Output Spooling Data Flow

steps. The starting block number of the new SYSOUT file is placed in the BLOCK entry by QLP, the STAT field is set to one (waiting), and then the CODE field must be set. For the left-hand system in Figure 3.3.1, the CODE field would be zero. However, the right-hand system must have a source code of zero, and a destination code indicating the left-hand system.

3.3.3 Spooling Initialization

After setting the output list, it remains only to initiate the line printer daemon. If ACTIVE is non-zero, LPD is currently "active." This means that the new entry in the output list will be processed by LPD after the currently active file and all other previously entered files are transfered. If ACTIVE is zero, LPD is idle, and needs to be awakened. In this case, QLP performs a subroutine call to the address specified in ADF.GO. This is the initiation sequence depicted in figure 3.3.2. ADF.GO points to procedure F.GO in LPD. F.GO first makes a special call to the line printer driver to enable the error trapping feature of that driver. Then the disk driver is called to read the first block of the listing file. After the disk read is initiated, F.GO exits back to QLP. Upon return from F.GO, QLP deletes any other SYSOUT files that are done. Whenever such deletions are made, the BLOCK numbers are set to zero. When these housekeeping duties are completed, QLP copies the output list to the STATUS.COM buffer in DLP, and then returns to DLP. DLP returns to MAINBA, as explained in section 3.1.0.

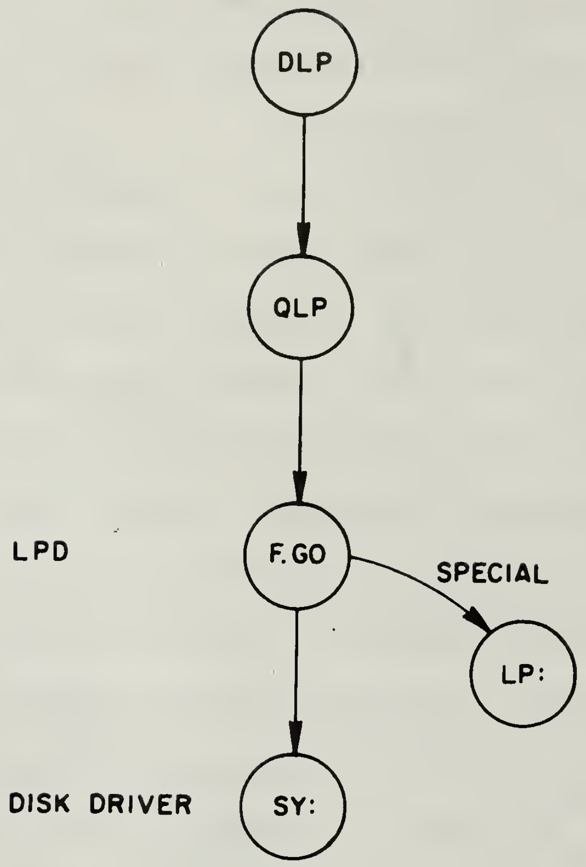


Figure 3.3.2

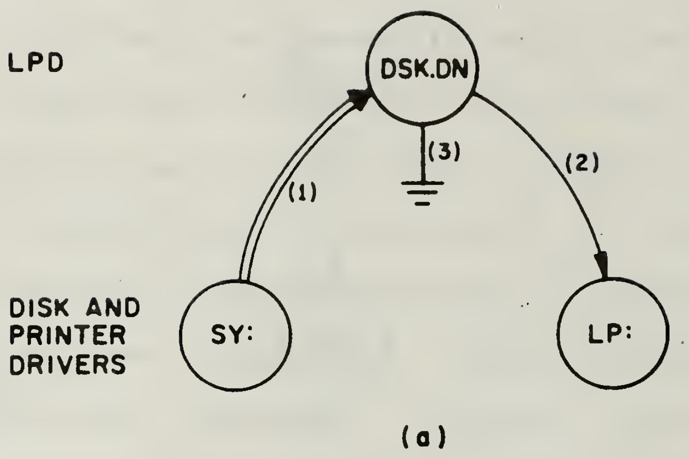
Line Printer Daemon Initiation

3.3.4 Print Process

The copying process in LPD is diagrammed in figure 3.3.3. When the disk driver takes its completion return (1), it interrupts LPD at DSK.DN (figure 3.3.3a). DSK.DN passes the disk buffer to the line printer driver (2) and exits from the disk interrupt (3). When the line printer driver takes its completion return (figure 3.3.3b), it interrupts LPD at LP.DUN (1). Disk buffers received at DSK.DN are 256 words long. The first word contains the disk block number of the next block to read from the disk; otherwise it is zero. Only words 2 through 255 are passed to the line printer driver. Therefore, when the completion return at LP.DUN is taken, we test the first word in the buffer. (Note, LPD is single-buffered; so, the disk buffer and printer buffer are the same.) If it is non-zero, we initiate the next disk transfer (2). If it is zero, the file transfer is complete. LPD then marks the appropriate STAT entry "done" in the output list, and searches the list for a "waiting" file. If a file is found, it is marked "busy," and the copying process continues; otherwise, LPD goes to sleep (3).

3.3.5 Line Printer Error Processing

If the line printer driver detects a device error during the copying process, it takes the special error return. This results in an invocation of procedure ERRET within LPD. ERRET uses the same error recovery algorithm as the card reader error processor in CRF; namely, we wait until the error condition disappears and then restart the I/O process. The interrupt to ERRET is represented by transition (1), figure 3.3.4a. The flow of



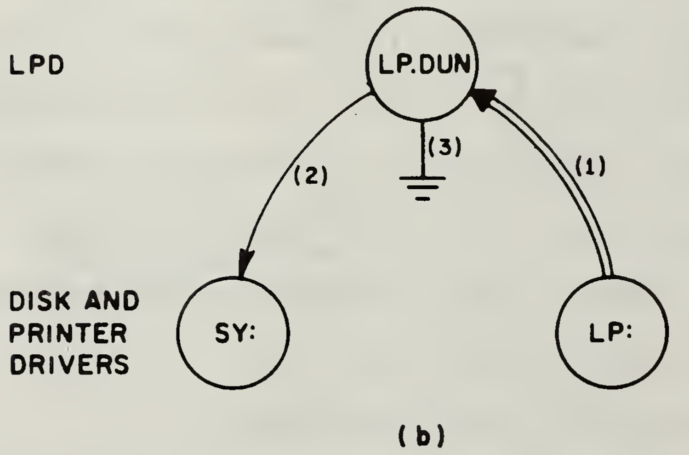
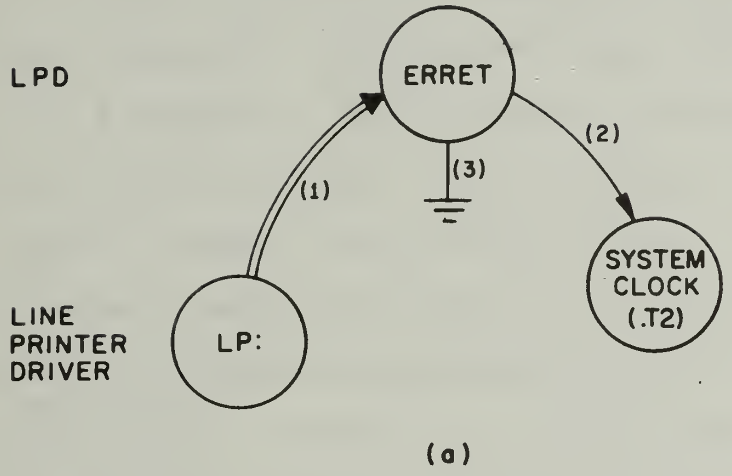


Figure 3.3.3

Line Printer Interrupt Processing



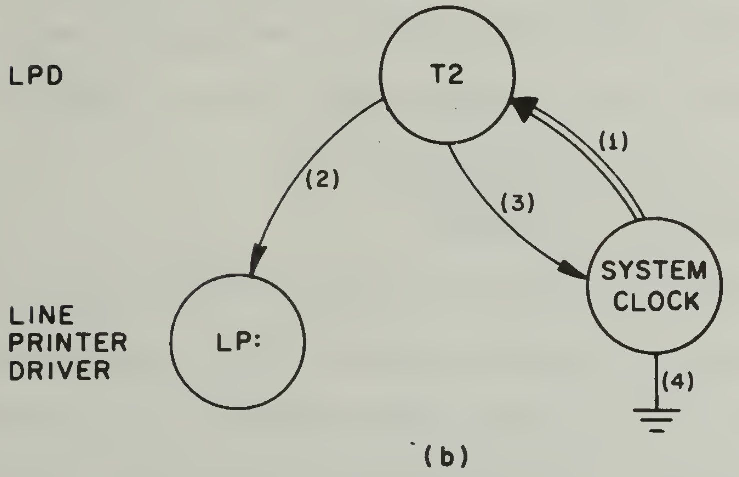


Figure 3.3.4

Printer Error Handling

control is by the numbers; i.e. (1), (2), and the interrupt exit at (3). Note that ERRET uses logical clock .T2, whereas the card reader routine uses logical clock .T1. Note also the different placement of ground symbols on figures 3.3.4a and figure 3.2.5a. This is a direct result of different implementations of the error trapping feature in the drivers as implemented by Atkinson and Stocks (see section 1.1). Once the waiting process begins, however, the line printer wait routine (figure 3.3.4b) is logically the same as the card reader routine (figure 3.2.5b) except that different variables are used for communication with the system clock routine. The control flow in figure 3.3.4b will be (1), (3), (4) as long as the error condition persists. When the error clears, the line printer driver will be restarted, and the control flow will be (1), (2), and (4). Processing should then continue with a completion return from the line printer driver as in figure 3.3.3b.

3.4 DOS/Multi-Batch Interface

Most interfaces to DOS follow standard DEC conventions and need not be given special mention here. We will discuss instead two non-standard and crucial interfaces: the Zip-file mechanism, and Multi-Batch's error-recovery mechanism. We begin with the processing sequence associated with Zip-files, as shown by figure 3.4.0.

BAT.V1 and BAT.V2 are two words in the resident DOS monitor which are not used by DOS. There is a unique constant that DOS normally places in these words which signifies the fact that they are an unused interrupt vector. When the DOS teletype driver

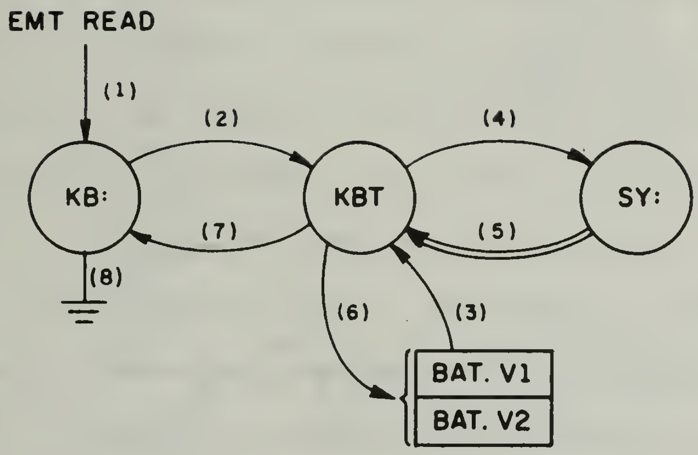


Figure 3.4.0

Zip-file Processing

(KB:) receives a read request (1), it first checks BAT.V1 to see if it contains the DOS constant. If the constant is found, then the driver operates in its normal mode by reading the console device. If some other non-zero constant is found, KB: calls a subroutine (KBT) which interprets BAT.V1 and BAT.V2 (3) as the physical disk address of a block of data, and an index into that block. KBT reads the disk (4) using this address, and waits for a completion interrupt (5) from the disk driver (SY:). The index into the disk block is used to find the next input line of text. This may require reading another disk block, although this sequence is not shown in the diagram. When a complete line of text has been assembled, BAT.V1 and BAT.V2 are updated (6), and KBT returns to KB: (7) where KB: executes its own completion return to DOS (indicated by the ground symbol).

If KB: finds a value of zero in BAT.V1, it means that Multi-Batch previously intercepted a DOS error trap (see below). When the zero is noted by KB:, the error-recovery scheme ensures that KB: always be processing a console read request from TMON, the DOS transient monitor program. TMON processes several DOS console commands including the RUN command. Thus, when the zero flag is found in BAT.V1, KBT passes a preset line of text to TMON, viz. "RUN BATCH." This initiates the foreground Multi-Batch monitor, which will also notice the zero flag (from routine START), reset it, and type an appropriate message.

The events associated with a zero indicator in BAT.V1 are diagrammed in figure 3.4.1. It happens that DOS uses the IOT instruction to indicate internal errors. This instruction causes

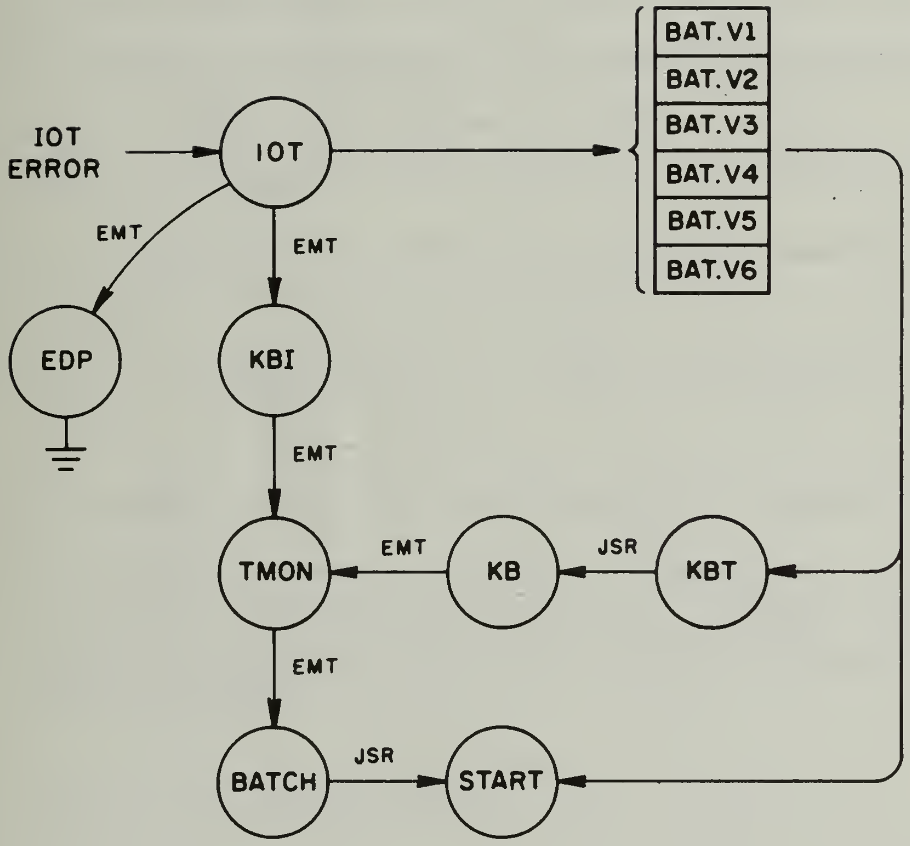


Figure 3.4.1

DOS Error Recovery

an interrupt-like trap through a fixed vector in low-core. This vector normally points to DOS's IOT service routine. In Multi-Batch, we change the IOT vector so that it points to our own IOT routine. It is this IOT processor which is depicted in figure 3.4.1. When IOT is entered on a DOS error, the error code is checked to see if the error is fatal or not. Non-fatal errors are passed to EDP since they will not interrupt Multi-Batch. On the other hand, fatal errors call for special processing. This consists of zeroing BAT.V1, saving the foreground program name and error codes in BAT.V3 through BAT.V6, and calling KBI, DOS's internal command interpreter. The IOT routine passes KBI a "kill" command. DOS terminates the current foreground job and initiates a thorough housecleaning sequence which is completed by TMON. TMON's first non-housecleaning act is to read a command from the console. However, BAT.V1 will still be zero, and TMON will receive a "RUN BATCH" command, as explained earlier.

LIST OF REFERENCES

[1] G. L. Chesson, "Communication and Control in an Experimental Computer Network," ACM 74 Proceedings, (1974) p.509-512

[2] IBM manual, OS/VS2 HASP II Version 4 System Programmer's guide, GC27-6992

[3] Burroughs Corp., Extended Algol Language Information Manual, 5000128

[4] Honeywell Corp., Multics Programmers Manual Volume IV - Subroutines, manual no. AG 93

[5] Digital Equipment Corp., RSX-11/M Internal Documentation (not published)

[6] IBM manual, IBM System/360 Operating System Job Control Language Reference, GC28-6704

[7] D. Kassel, "Batch User's Guide," informal memo, Department of Computer Science, University of Illinois, Urbana, Illinois (1974)

[8] J.L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," Computing Surveys, vol. 5, no. 1 (1973)

[9] J. L. Baer and E. C. Russell, "Preparation and Evaluation of Computer Programs for Parallel Processing Systems," in Parallel Processing Systems, Technologies, and Applications, edited by L. C. Hobbs, Spartan Mcmillan & Co. Ltd (1970)

[10] W. Bowie, A Distributed System for OS/360, Ph.D thesis, Univ. of Waterloo, 1974

[11] Data General Corp. RDOS Real time Disk Operating System Users Manual, manual no. 111-000075-04 (1973)

[12] Modcomp Corp. MAXNET III pre-release product specification no. 111-600305-000, (1974)

[13] K. Thompson and D. Ritchie, Unix Programmers's Manual, Bell Telephone Laboratories (1974)

[14] K. Thompson and D. Ritchie, "The UNIX Time-Sharing System," CACM, vol. 17, number 7 (1974)

[15] Hewlett-Packard Corp. HP-3000 Multiprogramming Executive Operating System, manual no. 0300-90005 (1973)

[16] S. Holmgren, The Network UNIX System, report no. 155, Center for Advanced Computation, University of Illinois, Urbana, Illinois (1975)

[17] S. C. Hsieh, Inter-Virtual Machine Communication under VM/370, IBM research report RC 5147 (1974)

[18] Per Brinch Hansen, RC-4000 Software Multiprogramming System, RCSLl no. 55-D140, A/S Regnecentralen, Copenhagen (1971)

[19] D.J. Farber and K.C. Larson, "The Structure of a Distributed Computing System - Software," in Proc. of the Symposium on Computer-Communications Networks and Teletraffic, Polytechnic Press, Polytechnic Institute of Brooklyn, New York, 1972, p.539-545

[20] Thomas, "A Resource Sharing Executive for the ARPANET," NCC (1973) p.155-163

[21] Burroughs Corp., Data Communications Extended Algol (DC Algol) Information Manual, 5000052

[22] J.B. Postel, Survey of Network Control Programs in the ARPA Computer Network, MITRE Corporation Technical Report MTR-6722 (1974)

[23] A. J. Bernstein, et. al, Process Control and Communication in a General Purpose Operating System, technical report 69-C-357, General Electric Research and Development Center, Schenectady, N. Y. (1969)

[24] Digital Equipment Corp., DEC System 10 Monitor Calls, manual no DEC-10-OMCMA-A-D

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-75-618	2.	3. Recipient's Accession No.
4. Title and Subtitle Multi-batch: A Multiprogrammed Multiprocessor Batch System for the PDP-11 Computer			5. Report Date April 1975 6.
7. Author(s) Gregory Lawrence Chesson			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Dept. of Computer Science University of Illinois Urbana, Illinois			10. Project/Task/Work Unit No. 11. Contract/Grant No. DCR72-03740A1
12. Sponsoring Organization Name and Address National Science Foundation Washington, D.C.			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

The internal organization of the multi-batch system is presented. The emphasis is on data and control structures, interprocess communication and control, and certain aspects of system development.

17. Key Words and Document Analysis. 17a. Descriptors

computer network, PDP-11, interprocess communication

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement release unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 87
	20. Security Class (This Page UNCLASSIFIED	22. Price